COMPLEXITY, HEURISTIC, AND SEARCH ANALYSIS

FOR THE GAMES OF CROSSINGS AND EPAMINONDAS

THESIS

David W. King Jr, Captain, USAF

AFIT-ENG-14-M-44

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

COMPLEXITY, HEURISTIC, AND SEARCH ANALYSIS

FOR THE GAMES OF CROSSINGS AND EPAMINONDAS

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyber Operations

David W. King Jr, B.S.C.S.

Captain, USAF

March 2014

AFIT-ENG-14-M-44

COMPLEXITY, HEURISTIC, AND SEARCH ANALYSIS

FOR THE GAMES OF CROSSINGS AND EPAMINONDAS

David W. King Jr, B.S.C.S.
Captain, USAF

Approved:

//signed//	4 Mar 2014
LTC Robert J. McTasney, PhD (Chairman)	Date
//signed//	4 Mar 2014
Maj Kennard R. Laviers, PhD (Member)	Date
//signed//	4 Mar 2014
Gilbert L. Peterson, PhD (Member)	Date

Abstract

Games provide fertile research domains for algorithmic research. Often, game research helps solve real-world problems through the testing and refinement of search algorithms in game domains. Other times, game research finds limits for certain algorithms. For example, the game of Go proved intractable for the Min-Max with Alpha-Beta pruning algorithm leading to the popularity of Monte-Carlo based search algorithms. Although effective in Go, and game domains once ruled by Alpha-Beta such as Lines of Action, Monte-Carlo methods appear to have limits too as they fall short in tactical domains such as Hex and Chess. In a continuation of this type of research, two new games, Crossings and Epaminondas, are presented, analyzed and used to test two Monte-Carlo based algorithms: Upper Confidence Bounds applied to Trees (UCT) and Heuristic Guided UCT (HUCT). Results indicate that heuristic knowledge can positively affect UCT's performance in the lower complexity domain of Crossings. However, both agents perform worse in the higher complexity domain of Epaminondas. This identifies Epaminondas as another domain that poses difficulties for Monte Carlo agents.

Para mi mariposa

Table of Contents

List of Figures

List of Tables

List of Acronyms

Acronym	Definition
MC	Monte-Carlo
AI	Artificial Intelligence
UCT	Upper Confidence Bounds applied to Trees
HUCT	Heuristic Guided UCT
LOA	Lines of Action
UCB	Upper Confidence Bound
RAVE	Rapid Action Value Estimation
MIA	Maastricht in Action
GGP	General Game Playing

COMPLEXITY, HEURISTIC, AND SEARCH ANALYSIS

FOR THE GAMES OF CROSSINGS AND EPAMINONDAS

I.   Introduction

Games provide test domains for Artificial Intelligence (AI) research and researchers often seek out new games to further algorithmic research in the community. Game research in the AI community often results in real-world applications of game theory in various environments. In addition, game research can identify search algorithm limits. For example, *Go* proved intractable for the commonly used Min-Max with Alpha-Beta pruning ($\alpha\beta$) algorithm, resulting in the introduction of Monte-Carlo (MC) based search [7]. Although highly popular today, even MC based algorithms appear to have limitations [9, 36]. Why do certain games, such as *Hex* and *Chess*, inhibit MC based search?

One proposed answer is that the commonly used Upper Confidence Bounds applied to Trees (UCT) algorithm is overly optimisitic in its move selection, resulting in a smaller exploration of the game tree [15]. Although Coquelin and Munos [15] modify the baseline UCT algorithm by cutting suboptimal branches from the search space, it has yet to gain traction in the AI community and warrants further investigation. This thesis extends this type of algorithmic analysis. In an effort to improve UCT's effectiveness, two MC based algorithms were tested across two new game domains: *Crossings* and *Epaminondas*.

Although created in the 1970's, *Crossings* and *Epaminondas* have escaped the community's notice. In order to understand where these games lie in the pantheon of currently researched games, agents for each game were constructed. These agents provided the information needed to derive the state-space and game-tree complexities of both games. The data indicates that *Crossings* has a slightly larger state-space and game-tree complexity

than the well researched game of *Lines of Action (LOA)* while *Epaminondas* provides a new testing domain between *Chess* and *Go*.

After construction of game playing agents for each game, these domains served as testing environments for UCT and a modified version of UCT called Heuristic Guided UCT (HUCT). The HUCT algorithm modifies the basic UCT formula by adding the heuristic value of the current board state to both the move's current win rate and Upper Confidence Bound (UCB) terms. Each algorithm plays against a baseline Min-Max $\alpha\beta$ agent with turns set to 1, 5, 10, and 15 second time intervals. Data indicates that adding heuristic knowledge increases the effectiveness of UCT in *Crossings* in both the 10 and 15 second categories. However, both UCT and HUCT performed poorly in *Epaminondas* across all time intervals. These results propose two main conclusions. One, MC based search agents perform well, and can even outperform Min-Max $\alpha\beta$ based agents, in *Crossings*. However, they do not perform well in the tightly related game of *Epmainondas*; identifying *Epaminondas* as another domain that confounds MC based search agents. The poor performance in *Epaminondas* may be due to the lack of a good heuristic evaluator, or that the combination of the game's complexity and tactical nature may lead MC agents towards bad parts of the search tree. Further investigation is necessary to understand why the MC agents struggled in *Epaminondas*.

1.1   Research Questions

The previous section introduced the basic premises of game research, the limitations of current algorithms in use today, and a brief overview of research into *Crossings* and *Epaminondas*. This section defines five specific research questions answered by the research presented.

1. How complex are *Crossings* and *Epaminondas*?

2. Do their unique movement rules impact their complexity?

3. Are *Crossings* and *Epaminondas* solvable?

4. Does adding heuristic knowledge to UCT improve its performance?

5. Does game complexity impact MC based algorithm performance?

These simple questions belie the complexity faced in answering them. First, answering questions one through three involves constructing Min-Max $\alpha\beta$ agents to play each game. Information to build heuristics to guide the search agent is sparse, residing in two main sources. Therefore, heuristic development relies on trying heuristics from similar games and strategies found through human game play. Refinement of baseline heuristics becomes imperative to achieve a novice level of play in order to answer all the questions presented. Questions four and five become answerable after these agents achieve a novice level of play since HUCT can then use the same heuristic function contained in the Min-Max $\alpha\beta$ agent to play each game. Comparison of UCT and HUCT performance across the domains relies on earlier work to establish the difference in complexities between *Crossings* and *Epaminondas*.

1.2 Impact

The research presented adds two new game domains to the AI field. Their unique moves lead to greater game complexity and provide two new research areas to test MC based algorithms. Furthermore, since research into these areas is brand new, deriving their state-space and game-tree complexities provides a categorization for both games. The discovery of MC failure in *Epaminondas* is noteworthy. It adds another domain to the AI field for future research and testing to help discover the underlying cause of such failures. Finally, all derived heuristics and saved game states provide starting points for any future work concerning either game.

1.3   Thesis Outline

Chapter II presents an overview of game study in Artificial Intelligence, motivation for game study, and the elements of game solving. In addition, it describes the most popular search algorithms in use today, and introduces the rules and basic strategies for *Crossings* and *Epaminondas*. Chapter III outlines the methodology used to answer each research question.   Chapter IV describes the design and development of the *Crossings* and *Epaminondas* game playing agents as well as descriptions of the experiments conducted. Chapter V presents results of those experiments and analyzes the collected data. Finally, Chapter VI presents conclusions drawn from the completed experiments and recommendations for future work.

II. Literature Review

Artificial Intelligence (AI) has a rich history of gaming research with applications extending beyond building game playing agents. Often breakthroughs in game research lead to real-world solutions. This chapter reviews the history of gaming research in Artifical Intelligence (Section 2.1) and why games are studied (Section 2.2). Section 2.3 discusses how games are played and *solved*. An overview of current search algorithms: Min-Max, Min-Max with Alpha-Beta ($\alpha\beta$) pruning, and Monte Carlo based search follows. Finally, Sections 2.9 - 2.10 provide the rules and basic strategies associated with *Crossings* and *Epaminondas*

2.1 Games in Artificial Intelligence

AI has a rich history of gaming research. Ever since Turing asked "can machines think?" researchers have sought to build machines capable of challenging, if not besting, human players [48]. Arthur Samuel took up Turing's challenge and constructed a *Checkers* playing agent in 1958 [41]. His groundbreaking work, while minimally successful, began a long tradition of researching games. Eventually, this led to Schaeffer *et al.* [43] solving the game of *Checkers* in 2007. The penultimate event for AI research seemed to occur when *Deep Blue* defeated the World *Chess* Champion Kasparov in 1997 [26]. However, defeating the World Champion did not usher in a new age of computer "thinking". Quite the contrary, researchers began pursuing domains where Min-Max $\alpha\beta$ techniques proved deficient [31]. This push in a new direction led to Monte-Carlo (MC) based agents. MC based agents excited the community because they needed nothing more than the legal moves of the game to be effective. It garnered attention when they produced agents that could play *Go* competently on 9 x 9 boards, eventually leading to agents playing on 19 x 19 boards at an amateur level [31].

The success of MC agents for the game of *Go* started a new conversation in AI research: can MC techniques work for other games where Min-Max $\alpha\beta$ is king? Can it best those agents? Or does MC suffer from some of the same drawbacks as Min-Max $\alpha\beta$ where, as the state-space and game-tree complexities grow, the effectiveness of the algorithm diminishes? On the heels of MC's success in *Go* the latter question seemed unlikely. However, games such as *Hex* and *Chess* remain elusive to MC methods. Is there something more to these games other than their complexities that hurts MC based search?

The goal of analyzing the domains of two closely related games: *Crossings* and *Epaminondas*, is to help shed light on this question. If these games prove difficult for MC methods to play, what makes them special? Is there something more to these games?

2.2 Game Study

Why do researchers spend so much time studying games? One can claim that games and human culture are intertwined. The oldest gaming pieces found date from 5,000 years ago [34]. Every society and culture plays games. From *Go* in China and *Shogi* in Japan, *Chess* worldwide, *Senet* and *Seega* in Egypt, *Pachisi* in India, *Mancala* in Africa, and one of the oldest games called *Ur* found in Persia [8]. Games play a significant role in human society and studying games can help researchers understand human cognition better. They also present one of the few domains where machine knowledge can be directly tested, and measured, against humans. Outside of being a part of human culture, games help model and solve real world problems.

For example, game theory concepts derived from strategy games is applied to help combat the increase of vehicular traffic in urban areas [25]. Researchers model normal traffic patterns and how road closures and construction affect traffic as games. Solutions to these games guide traffic policy and decisions [25]. In another example, medical practitioners use game theory to develop an understanding of patient trust with respect to medical care [45]. This is also an example of understanding human thought as the idea

6

of *trust* is scrutinized and digested to assist medical practitioners in putting their patients at ease. The AI community also uses the game domain as a test bed for algorithmic research.

As complex games came under study, the algorithms used by AI researchers to solve them became more complex. Algorithms such as MinMax [38], MinMax Alpha-Beta [29], Upper Confidence Bounds applied to Trees (UCT) [31], *etc.*, were tuned and modified to handle playing games. These modifications led to breakthroughs in how humans played games. Tesauro's TD_Gammon [46] agent brought a revolution in *Backgammon* as a once eschewed opening proved to be very strong in tournament play . His program changed how humans played *Backgammon* at a professional level. The development of MC based search algorithms is a direct result of previous algorithms failing to gain any momentum in playing *Go*.

Finally, games are fun, but often difficult to program. A key interest in MC algorithms is the relative ease of implementation. A programmer can avoid coding in complex strategies. Games such as *Go* suffer from this as long-term strategy for *Go* is very subtle and difficult to grasp, let alone program into an agent [7]. Creating an agent to play competently is difficult. Researchers often modify games to gain traction on the problem. At times, reducing the board size can make an unsolvable game solvable. For example, Winands shrank the game *Lines of Action* from 8x8 to 6x6 and proved it solvable on the smaller board [53]. Researchers solved versions of *Hex* on boards up to 8x8 in a similar manner [37].

2.3 Algorithm Development and Popular Games

One of the more popular, knowledge based, search algorithms is Min-Max with Alpha-Beta ($\alpha\beta$) pruning [29]. As a knowledge based search algorithm, a heuristic function guides an agent towards promising moves by pruning the search space. Usually, Min-Max $\alpha\beta$ is configured to search to a predefined cutoff depth, meaning it searches and returns the node that appears to have the highest likelihood of success versus the best move. As the

state-space and game-tree complexities expand, the more likely it becomes that the move returned is the not the best one. This is called the *horizon effect* and occurs because the algorithm has to cutoff its search at a certain depth *(d)* due to time constraints [38]. Many levels of the game can exist below this level. A move at level *(d)* may appear good when, in reality, at level *(d+1)*, it is a game loss. Researchers often modify Min-Max $\alpha\beta$ to minimize these issues. These enhancements cut down the amount of search space the algorithm sifts through, enabling better move returns. Other variations have the agent select a move and then do a small two to three turn look ahead from that move to counter the horizon effect. However, there is still a limit. In addition, Min-Max $\alpha\beta$ relies heavily on encoded heuristic knowledge. If the heuristic function overestimates the strength of the player's position, then the agent can prune away winning lines of play or mistake losing plays for winning ones. This is particularly true in the case of *Go*.

Go's search space is the largest for a game of perfect information that researchers are trying to tackle today [3]. Until the early 1990s, many felt that *Go* was unsolvable. In 1993, Brügmann applied the idea of simulated annealing to his *Go* playing agent *Gobble* obtaining remarkable results [11]. Instead of giving the game all the knowledge of the coder for *Go*, he let the agent play out as many lines of play as it could in a set timeframe. Once time expired, the agent played the best line it found. The connection to multi-armed bandit problem solving in this approach is evident [38]. Over the next decade, researchers modified his approach and developed MC agents that played *Go* well on 9x9 boards [18]. The main strength of MC agents is they do not need any game knowledge to play a game. This is a huge benefit to researchers. However, although MC agents are simpler to code, researchers have yet to beat human players on 19x19 *Go* boards above an amateur level [31]. *Go* research also outlines a common approach to game research; testing new techniques and modifications on smaller boards and then extending them to larger ones. If

an algorithm proves successful, researchers apply it to a larger version of the game. This holds true for games outside of *Go*.

Winands' work with *Lines of Action* (LOA) began on the regular sized LOA board for his Master's and leveraged that research for his PhD [51, 52]. His agent, Maastricht in Action (MIA), placed highly at the Computer Olympiad and progressed to one of the best LOA playing agents in the world. He modified Allis' PN search, the standard Min-Max $\alpha\beta$ search, as well as hybridizing a MC agent, creating his own algorithm called Monte-Carlo Tree Solver [54, 55, 57–59]. He also shrank the board, and solved LOA on a 6x6 board [53]. He used both LOA versions as a test beds for algorithmic development, analysis and testing. *Hex* is another example of such work.

John Nash reinvented *Hex* while at Princeton in 1948 [37]. Although a simple connection game, the state-space and game-tree complexity grow exponentially as the size of the board increases. With its simple rules and expandability, *Hex* provided fertile ground for algorithmic research. Researchers have solved smaller versions of *Hex*, but boards over 9x9 remain elusive [23, 37]. *Hex* is interesting on two counts: one, there is never a draw, and two, it is *weakly solved*. This means there is a strategy that always wins. In the case of *Hex*, the first player can theoretically never lose. The argument goes that if the second player employs a winning strategy, then the first person can steal it and use it to win [37]. Research of this type continues today with researchers looking for other games to explore and analyze.

2.4   Solving Games

Go is one of the most complex games researched today and researchers continuously strive to develop agents that can play at a professional level with efforts using various techniques such as Abstract Proof Search [12], Lambda-Search [47], and Monte-Carlo Tree Search (MCTS) [17, 40]. The main goal for a game researcher is *solving* the game at hand. Solving a game means finding the game-theoretic value of a given position [40]. This value

indicates who will win the game [24]. There are three categories for solving a game: ultra-weakly solved, weakly solved, and strongly solved [3]. In ultra-weakly solved games, the game theoretic value has been determined for the initial board state. For weakly solved games, a strategy has been determined to obtain the game-theoretic value of the game for both players. Finally, strongly solved games are those where an agent has a strategy or game-theoretic value for all legal positions. With a new game, determining its solvability is often a researchers first task.

Herik, *et al.* [24], developed four categories to help researchers determine the solvability of a game. The game's state-space and game-tree complexities play a vital role in determining its category. Games with low state-space and low game-tree complexity are easily solvable, usually through enumeration of all moves (brute-force) or a basic, algorithmic strategy that always leads to a win or draw. *Tic-Tac-Toe* is an example of such a simple game. Brute force methods can solve games with a low state-space but high game-tree complexity. *Nine-Men's Morris* and *Checkers* fall into this category of games. The current upper bound for solving games via brute-force methods is approximately 10^{20}. Schaeffer derived this bound while solving *Checkers* although, in reality, he reduced the state-space to 10^{14} through the elimination of illegal states and adding move prioritization [43, 44]. Herik, *et al*., further solidified the 10^{20} bound in [24]. Knowledge-based methods can solve games with a high state-space but low game-tree complexity. In these games, researchers introduce game knowledge to reduce the search space allowing the agent to find the best move available for the current position. *Go-Moku* [4] and *Renju* [49] are examples of such games. Finally, *unsolvable* games possess both high state-space and game-tree complexities where all known methods fail to solve them. The best examples of these types of games are *Hex*, *Chess* and *Go*. For such games, researchers usually reduce the board size to make them solvable as is the case of *Hex 8x8*, *Go 5x5*, and *Lines of Action 6x6* [23, 50, 53]. In order to determine a game's solvability, a researcher must derive a

game's complexity. In order to do that, the researcher must build an agent to play it. This task starts with selecting an appropriate search algorithm.

2.5 Min-Max

Researchers want to build agents that make optimum decision at every phase of the game. The Min-Max algorithm returns the optimal decision from the current game state [38]. It computes the min-max values for every reachable state in the tree and then returns the move that leads to the optimal state. The algorithm works in a depth first recursive manner where it moves down one branch to a leaf node, then recurses back up to go down another branch and so forth. Eventually, the algorithm visits *every* reachable state. Although it finds the optimal move, the time cost quickly becomes intractable for large game trees since the complexity for the algorithm is $O(b^m)$ where m is the maximum depth and b is the legal moves at each point [38]. The algorithm is easy to implement but finds minimum use beyond trivial games such as *Tic-Tac-Toe*. *Chess* is a prime example of this issue.

The game-tree complexity for *Chess* is approximately 10^{123} with an average game length of 80 and a branching factor of 35. If a Min-Max agent computes one million moves a second, it would take an astonishing $3.35 * 10^{109}$ years to return the optimum move from the initial board position. See Equation 3.1.

Min-Max Solving Chess Equation (2.1):

$$\frac{O(b^m)}{3.15^{14}} = \frac{35^{80}}{3.15^{14}} = 3.35 * 10^{109} \tag{2.1}$$

This timeline only gets worse as the game complexity increases. With large game-tree complexities, a researcher must use an alternative to min-max. In 1975, Knuth developed the Min-Max Alpha-Beta ($\alpha\beta$) algorithm that prunes parts of the min-max tree that hold values above or below a certain threshold [29]. In this manner, one can reduce the search

space, saving time and enabling an agent to return an answer in a reasonable amount of time.

2.6   Min-Max with Alpha-Beta Pruning

The basic premise of Min-Max with Alpha-Beta ($\alpha\beta$) pruning is to assume that one player, *max*, will always play the move that *maximizes* their position, while the second player, *min*, will always choose the move that *minimizes max's* position [21]. Beginning at the root of the tree, *max* begins a search bounded by a depth *d*. At level zero, there is a max node, then level d+1, a min node, at d+2, a max node, and so forth. Each level switches the player's perspective as each side takes future turns. Once a terminal node, or the depth is reached, the algorithm evaluates the position from the perspective of whose turn it is: *max* or *min*. It then backtracks one step, forwarding the value upwards. It then proceeds down the next branch. If a value for a subtree is encountered that is higher for a minimum node, or lower than a maximum node, the search stops looking at that subtree, effectively cutting it off. This allows the algorithm to prune the space, reducing the time needed to find a solution. Furthermore, setting a bound for the search also allows the agent to quickly make decisions. Responsiveness is important in games. Humans are not overly patient creatures, preferring an agent that can return a move in under a minute for games such as *Checkers* and *Lines of Action*. Researchers usually extend these time limitations for games such as *Chess* and *Go* where players often take minutes to make moves. Algorithms 1 and 2 present the pseudocode for the Negamax version of Min-Max $\alpha\beta$ [27]. Figures 2.1 and 2.2 show how Min-Max $\alpha\beta$ differs from the regular Min-Max algorithm as an agent traverses the same search space.

Algorithm 1 min_max()

1: **if** *TerminalPosition* **then**
2:     **return** *h_value*()
3: **end if**
4: moves = createChildren()
5: moves = orderMoves(moves)
6: best_move = moves
7: **for all** *children* in *moves* **do**
8:     makeMove(child)
9:     oppMove = min_max()
10:    val = -oppMove.value
11:    **if** *val > best_move.value* **then**
12:        best_move = child
13:        best_move.value = val
14:    **end if**
15:    reverseMove(child)
16: **end for**
17: **return** *best_move*

Algorithm 2 Negamax – alphaBetaMinMax(depth, alpha, beta)

___

if *depth* ≤ 0 **or** *TerminalPosition* **then**

    return *h_value*(*move*)

end if

moves = createChildren(move)

moves = orderMoves(moves)

best_move = moves

for all *children* in *moves* **do**

  if *best_move* ≥ *beta* **then**

    return *best_move*

  end if

  makeMove(child)

  if *alpha* < *best_move.value* **then**

    alpha = best_move.value

  end if

  opponentMove = alphaBetaMinMax(depth-1, beta, alpha)

  oppVal = -opponentMove.value

  if *oppVal* > *alpha* **then**

    alpha = oppVal

    best_move = child

  end if

  reverseMove(child)

end for

return *best_move*

___
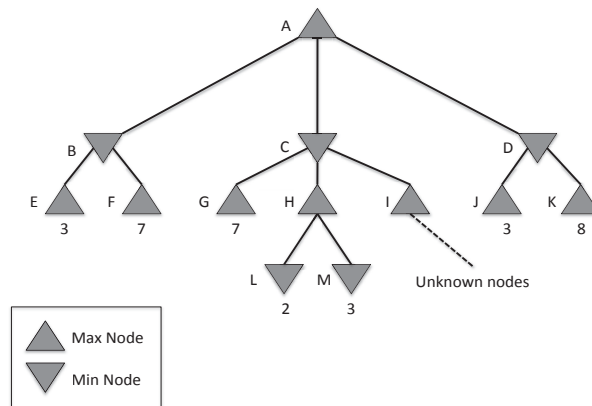
Figure 2.1: Min-Max
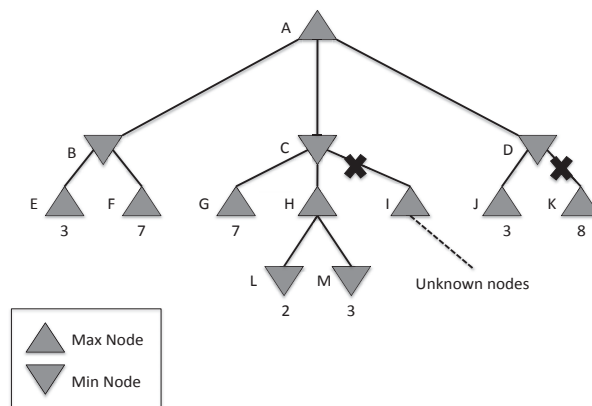
Figure 2.2: Min-Max with Alpha Beta Pruning

In Figure 2.1, the Min-Max algorithm will explore every single node, saving the best max and min values at each node. The Min-Max agent will terminate only once it explores

the entire tree. In this manner, Min-Max returns the optimum move. However, as discussed earlier, Min-Max may not terminate in a reasonable amount of time. Although complete, Min-Max is not very useful beyond small board states.

A Min-Max $\alpha\beta$ agent can reduce the search space significantly. In Figure 2.2, one can see that parts of the tree are pruned when values go beyond the current $\alpha\beta$ cutoffs. These cutoffs can save tremendous amounts of computational time. In both figures, there are *unknown nodes* underneath the *I* node. This subtree can be of an arbitrary size but Min-Max $\alpha\beta$ trims the *I* node. The algorithm can do this since, from earlier exploration, node C, a *min* node, will not allow the agent to select nodes whose value is beyond three. Since it has already hit this value, the algorithm can quit its search at this branch and recurse back to the root node, A. Min-Max does not do this, and will explore all of these *unknown nodes*, slowing the agent down. With pruning, Min-Max $\alpha\beta$ can search a tree twice as deep in the same timeframe as a typical Min-Max search [38]. Enabling deeper and quicker searches allows agents to make better choices for their particular environments. However, because of the horizon effect discussed earlier, bounded Min-Max $\alpha\beta$ agents may not return the optimum move. The algorithm returns a move that is *estimated* to be the best one. This is the main trade off when using a bounded Min-Max $\alpha\beta$ agent.

2.7   Alpha-Beta Enhancements

There are four major enhancements for Min-Max $\alpha\beta$ search: move ordering, killer moves, history heuristic, and transposition tables. Although Schaeffer [42] debates if all these enhancements are effective, most AI researchers implement them in their Min-Max $\alpha\beta$ agents.

2.7.1   Move Ordering.

Min-Max $\alpha\beta$ relies on good move ordering to be effective [35]. The main idea behind move ordering is to have the algorithm look at *good* moves first. Schaeffer best defines *good* moves as either ones that cause a cutoff or the one that yields the best min-max value

[42]. In this manner, cutoffs occur quickly and prune large areas of the tree. This has a two-fold effect: one, it speeds up the search, and two, the agent is more likely to select winning moves since it can continue to search valid parts of the tree without wasting computational time on *bad* moves. If one avoids using this technique, then, in the worst case, Min-Max $\alpha\beta$ will search all the moves at each level before finding the best move. This worst case scenario forces Min-Max $\alpha\beta$ to run in $O(b^{\frac{3m}{4}})$ while, with proper move ordering, this can be reduced to $O(b^{\frac{m}{2}})$ where b is the branching factor and m is the maximum depth of the tree [38]. Move ordering significantly reduces state-space exploration allowing a Min-Max $\alpha\beta$ agent to look deeper into the tree, in less time, than regular Min-Max.

2.7.2 Killer Moves.

In the original Min-Max $\alpha\beta$ method, once an agent returns a move, it scraps all state evaluations. Every time the agent needs to make a move, it has to rediscover cut-off values that, in all likelihood, are close to, if not the same, as the prior search since board positions do not change dramatically from move to move. Instead of throwing out the old cut-off values, the agent saves moves that caused cutoffs but were not the move selected for play. When a new search begins, the agent retrieves these killer moves and, if valid, uses them in the current position to expedite the search [42]. This heuristic saves a killer move for each level of the search that produced a cut off [35]. Trying these moves first helps eliminate parts of the tree, thereby, increasing the effectiveness of Min-Max $\alpha\beta$ searches since the agent is pruning the tree without having to calculate new cut-off values. Min-Max $\alpha\beta$'s iterative search behavior enables the use of this technique.

2.7.3 History Heuristic.

The history heuristic is a general case of Killer Moves [42]. Instead of saving only a handful of moves, the history heuristic saves the success rates for all moves at all depths. After move generation, the agent orders moves based on their history scores, leading to $\alpha\beta$ cutoffs [35]. Over time, the history value is reduced since the game is progressing away

17

from those moves, *i.e.* their impact on the game state fades as moves are made. Again, this enhancement reduces the space for the agent by cutting parts of the search tree.

2.7.4   Transposition Tables.

Transposition tables [42] reduce recalculation of states significantly.   Instead of throwing out evaluated states, the Min-Max $\alpha\beta$ agent saves them in memory. Transposition tables save information about the value of a subtree, the move that led to that tree and its depth . When the agent encounters a state, it queries the transposition table first. If the state is in the table, the query will return its value. Otherwise, the agent evaluates the state normally and saves it to the transposition table along with its depth. Saving states in memory saves computation time. Normally, researchers implement transposition tables as hash tables. The advantage of hash tables is quick lookups. The average look-up time for an element in a hash table is $O(1)$ [16]. Insertion and deletion operations are also $O(1)$ operations. Hash tables have two limiting factors: the hashing function and memory requirements.

A complicated hash function will slow down hash table operations. Zobrist Hashing [62] is a simple and effective hash function. Zobrist hashing uses simple XOR-ing of the board state, with other data such as its depth in the tree, to produce an index. In order to retrieve the element, one just needs to XOR the current move and depth to produce the key for look up. Zobrist hashing is a very simple, elegant, and most importantly, fast, way to store and retrieve data from the hash table.

The second issue with hashing is memory space limitations. Since memory is finite, one has to maintain a hash table size that is smaller than the number of reachable states. Inevitably, since the key space is smaller than the state space, collisions will occur. There are a number of ways to deal with hash collisions. One can keep the old value, dispensing with the new, or keep the new and dispense with the old, or, chain the objects together, basically forming a linked list off of the hash index [16]. In the case of a collision, the

implemented Min-Max $\alpha\beta$ agent replaced the older hash table object with the latest one under the assumption that the agent is unlikely to revisit the older state in the current game.

The Min-Max $\alpha\beta$ algorithm is the most common search algorithm used for game agents today. It can quickly create an agent to play a game and through heuristic refinement, the agents can play up to a professional level in some games. However, Min-Max $\alpha\beta$ never played above a novice level in the game of *Go* on small boards leading researchers to look elsewhere for answers.

2.8 Monte-Carlo Based Search Methods

Monte-Carlo (MC) based search methods have garnered interest ever since their breakthrough in competently playing *Go* on small boards. The majority of *Go* programs today use MC based search algorithms [10]. In their short article on MC methods, Lee *et al.* [31] outline a quick history of MC based search algorithms and their impact on *Go* research. Brown *et al.* provided an in-depth survey of MC methods in [10]. MC based search began with Abramson's idea of averaging the results of simulated random games from the current board state [31]. In 1993, Brügmann applied the idea of simulated annealing in his *Go* playing agent *Gobble* obtaining remarkable results [11]. Although Brügmann appeared to make a breakthrough in *Go*, his work went relatively unnoticed for about a decade. In 2002, Bouzy *et al.* [7] successfully applied MC methods to 9x9 *Go* with their program *Olga* and *Oleg* by editing the simulated annealing portion of Brügmann's work to fit Abramson's original work. More pieces to the puzzle fell into place with Coulum's *Go* agent *Crazystone* [17]. Coulum added a stricter discriminatory selection of played out nodes. His algorithm selectively chose the move to play out versus using a randomly selected move. This improved *Crazystone*'s performance, allowing it to beat many of the *Go* agents at that time [17]. Eventually, this algorithm matured with the introduction of Upper Confidence Bounds Applied to Trees (UCT) to influence move selection [30]. Winands *et al.* pushed MC based search further by adding Min-Max $\alpha\beta$ selected play-outs to increase the agent's

probability of selecting *good* moves [55–58]. These enhancements enabled the MC agent to defeat Winands' highly successful *Maastricht in Action (MIA) Lines of Action* (LOA) agent. Currently, MC based *Go* agents play at a professional level for 9 x 9 boards and amateur levels on 19 x 19 boards thanks to MC breakthroughs [31]. MC has also shown success in games such as *Hex*, although Browne showed some board states remain elusive [5, 9]. However, it appears to fail in its application to *Chess* [10, 36].

The basic MC based search algorithm consists of four parts: selection, expansion, play out, and back propagation. Figure 2.3 shows one iteration of a basic MC based search algorithm [10].
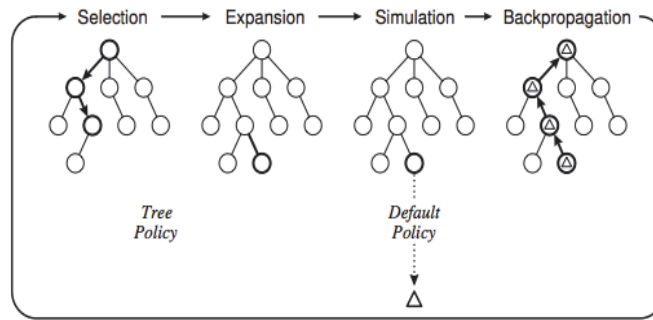
Figure 2.3: MC Based Search – One Iteration [10].

In the selection portion, the agent selects a node for expansion. An agent may select an action randomly, the basic MC version, or it may select an action guided by a user-defined policy such as UCT. After selection, the agent expands a node and adds the node's children to the search tree. Once added, the agent performs a play out to find a value for the selected node. A play out may consist of a series of random moves, or guided in some manner

20

(see [56]). This defines the *default policy* for the selected algorithm. Once a play out is complete, its value is backpropagated up the search tree and the process begins anew.

One of the major benefits to MC based search algorithms is they do not require a heuristic function to evaluate the board state [10]. They only require the rules and logic to determine a win, loss, or draw. In its simplest form, a MC based search agent generates moves from a given state, randomly selects one, stores it in memory, and then plays a game from that point. The agent selects random moves at each level until the game ends. The agent evaluates the final board as a win (+1), loss (-1) or draw (0). This value is then backpropogated up the tree to the original selected node. The agent then begins again by selecting a move at random, which could be the same node or a different one, and continues until time expires. Once time is up, the agent returns the child node with the highest win percentage. The completely random MC based search agent usually performs poorly since there are no guarantees that it will explore winning moves over losing ones. In addition, the agent randomly selects moves in the play out step, versus selecting moves that may be beneficial. To counter this, most researchers implement UCT.

2.8.1 Upper Confidence Bounds Applied to Trees.

In UCT, upper confidence bounds (UCB) guide the selection of a node, treating selection as a multi-armed bandit problem [10]. Equation 2.3 shows the formula for UCT node selection:

UCT Evaluation Equation (3.3):

$$Value = X_j + C * \sqrt{\frac{\ln{(n)}}{n_j}} \tag{2.2}$$

X_j is the win ratio of the current state, n is the number of times the parent state has been visited, n_j is the number of times the current state has been visited, and C is a constant between 0 and 1 where higher values and lower values adjust the amount of exploration done by the agent [10]. UCT theoretically converges to Min-Max if given infinite time and memory and thus is optimal in that scenario. Browne showed the advantage of UCT

21

over the pure random MC based search agent in *Hex*. While the random MC agent failed to solve relatively simple, tactical positions in *Hex*, UCT correctly solved them [9]. UCT served as the baseline MC agent for *Crossings* and *Epaminondas*.

2.9 Monte Carlo Enhancements

The main divergences in MC based search implementations reside in the selection and play out stages. Winands proposes that constructing a smaller, heuristic guided search tree with Min-Max $\alpha\beta$ produces better results in *Lines of Action* [56]. There is a price to tweaking selection and play outs. Any modification to the selection or play out stages reduces the number of nodes selected for play out. Furthermore, it reduces the number of times the algorithm runs in the specified time. Both of these factors impact the effectiveness of the MC based agent. In essence, Winands' method gambles that the Min-Max $\alpha\beta$ based play out derives correct values for all nodes. In this case, all nodes selected for expansion will contain values close to those a Min-Max $\alpha\beta$ agent could produce. The main issue with this technique is the Min-Max $\alpha\beta$ portion of the agent must be highly tuned to achieve such results. If the agent incorrectly estimates the node value, then the agent will not only explore a minor portion of the tree space, it will explore a *bad* part of the tree, resulting in poor play. A popular alternative to UCT is Rapid Action Value Estimation (RAVE).

2.9.1 Rapid Action Value Estimation.

RAVE differs from UCT in two ways. First, RAVE adds another value estimate to the Upper Confidence Bound (UCB) in UCT, shown in equation 3.2:

RAVE State EvaluationEquation (2.3):

$$StateValue = X_j * \beta + (1 - \beta) * \sqrt{\frac{\ln(n)}{n_j}} \tag{2.3}$$

where β equals $\sqrt{\frac{k}{(3*numGames+k)}}$. The β parameter tempers the impact of UCB as well as the win rate estimate. The more the agent plays out a move, the more weight the moves win rate holds. Researchers derive k through testing. Secondly, RAVE updates

22

any moves encountered during random play out that currently exist in the search tree, with the value found at the end of the game. This is the equivalent of playing out those nodes simultaneously. Cazenave [13] and Browne [9] provide a thorough treatment of RAVE. RAVE enabled MC based agents to play *Go* at a professional level on 9x9 boards and an amateur level on 19x19 boards [31]. RAVE still proves inadequate for *Chess* and adds complications to the backpropagation step since encountered nodes during play out must be cross-referenced with any nodes expanded in the search tree. Furthermore, RAVE ignores heuristic guidance although Winands shows that heuristic knowledge can impact the success of MC based search agents [56]. However, Winands' method is complicated to implement and requires a highly tuned heuristic evaluation function to work properly.

2.9.2 *Heuristic Guided UCT.*

The idea behind the Heuristic Guided UCT (HUCT) approach is to leverage strong game evaluators built for Min-Max $\alpha\beta$ to overcome the difficulties MC agents have in tactical domains. Lorentz and Horey [33] use a heuristic evaluation function of the board state to backpropagate a win or loss in limited MC roll outs in their *Breakthrough* agent. After this modification, their agent outplayed the majority of Min-Max $\alpha\beta$ agents on a popular game server. In earlier work, Winands' heuristic guided UCT achieved balanced play against his LOA playing agent Maastricht in Action (MIA) [57, 58]. Researchers achieved similar results in *Amazons* [28, 33] and *Go* [19].

The drawback to the heuristic guided approach is the agent must perform the move in order to evaluate it. This means added computation time, resulting in fewer simulations per time interval. As noted, fewer simulations can result in poor play. Furthermore, the heuristic value weighs heavily in node evaluation. If the heuristic is poor, the algorithm may suffer. However, research indicates that HUCT usually outperforms the basic UCT algorithm and can play on par with some Min-Max $\alpha\beta$ agents. Chapter 5 provides a

comparison of UCT and HUCT implementations playing against Min-Max $\alpha\beta$ agents in *Crossings* and *Epaminondas*.

2.9.3   Threading.

Researchers have shown that threading MC based search agents can be a relatively easy task and can increase the performance of the agent. Yoshimoto *et al.* proved there is a point of diminishing returns for threaded MC based search agents [60]. There are three types of threading possible: root, leaf, and tree [14]. The simplest is root threading. Here the agent launches separate MC based threads with the same root node. Once time is up, the parent thread returns the best move from the thread returns. The benefits of this method is it is the simplest to implement and there are no shared memory issues.

A secondary method is leaf parallelization. Here, after the agent selects a leaf node to play out, it spawns multiple threads from that move (each represents a simulated game play out from that node). If one is optimistic, the agent assigns the highest value returned to the move. Since MC agents often underestimate the value of a position, this is prudent [56]. However, one can err on the side of caution and assign the lowest value returned as well.

Finally, Chaslot *et al.* [14] introduced the tree parallelization method. In their algorithm, all threads have access to the search tree. The threads run simultaneous games at once, sharing information as they play out. Here one must maintain global and local mutexes to avoid corruption of the search tree. As threads complete their simulations, the agent backs up the values to the shared tree. The major downside to this type of threading is the introduction of mutexes that are notoriously hard to implement correctly and troubleshoot. Additionally, Chaslot *et al* failed to prove any true benefit from implementing this type of threaded MC based search agent. Additionally, the implementation seems very similar to the RAVE technique with more coding overhead involved.

2.10 Rules and Strategies for Crossings and Epaminondas

Literature on *Crossings* and *Epaminondas* is sparse with the main sources coming from the rules published in Sackson's book, *A Gamut of Games*, as well as Abbott's expansion of the game into *Epaminondas* discussed in an article for *Abstract Games*, as well as his own article about the games [1, 22, 39]. *Crossings* and *Epaminondas* are zero-sum, two-person strategy games with perfect information. Abbott invented *Crossings* in the late 1960's [39]. A few years later, Abbot increased the complexity of *Crossings* by expanding the board to 12 x 14 and modified the capturing rules to place more emphasis on flanking maneuvers [1]. He dubbed his new creation *Epaminondas* [22]. The basic premise of both games is to move one's pieces to the opponent's back row. Once a piece lands on an opponent's back row, a move called a "crossing," the attacked player has one turn to respond. Here the two games diverge. In *Crossings*, the only response for the attacked player is to complete their own *crossing*. In *Epaminondas*, the attacked player can either complete a *crossing* or capture the offending piece. If these conditions are met, the game continues. Otherwise, the game is over, and the player who made the last *crossing* wins. This creates a unique gaming experience steeped in forward-thinking strategy.

2.11 Crossings

The following sections outline the rules and strategies for *Crossings* [39].

2.11.1 Overview.

Crossings is played on a 8 x 8 checkered board with 16 white and 16 black pieces. The initial starting position is displayed in Figure 2.4. White moves first, followed by black and so on. A player cannot pass. The goal is to move a piece onto an opponent's back row. Whoever has more pieces on their opponent's back row after one full turn, is the winner.
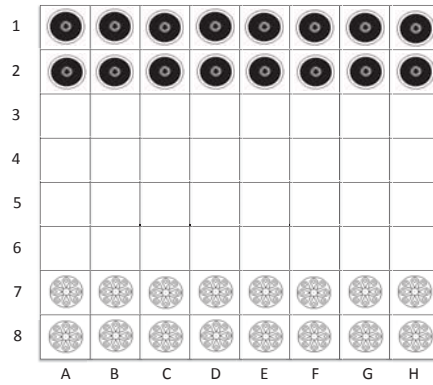
Figure 2.4: Crossings Initial Position.

2.11.2 Phalanxes and Movement.

One or more pieces can move at a time. Only pieces adjacent in a straight line can move together. Singleton pieces can move in all directions. Phalanxes can only move along the line in which they are orientated. The maximum number of squares a phalanx can move is equal to the number of members of the phalanx. For example, a phalanx of two can move one or two squares. A phalanx of one can only move one square. Subphalanxes of a larger phalanx may move independently. In other words, one chooses how many members of a phalanx will move and how far the phalanx will travel up to the max allowable distance. See Figure 2.5.
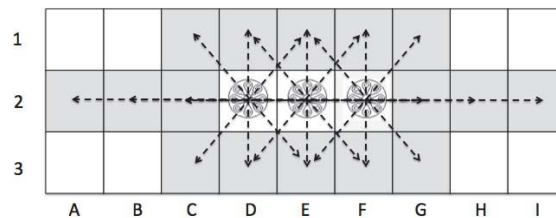
Figure 2.5: Example Phalanx Moves.

26

2.11.3 Capturing.

When a phalanx of two or more runs into an enemy phalanx that is strictly *smaller* than it, the first encountered piece may be captured. The attacker captures only the lead piece and the phalanx stops on that square. See Figure 2.6. If a phalanx encounters an enemy phalanx of equal size, then movement halts in front of the enemy phalanx. In Figure 2.6 if White had a piece at I2, then Black's phalanx would have been unable to capture White's piece at F2.
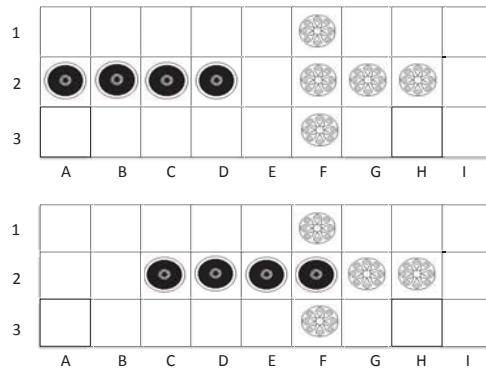
Figure 2.6: Crossings Capture Example.

2.11.4 Objective.

Once a player moves a piece to the opposing back row, a *crossing* has occurred. Unless an opponent responds immediately with their own *crossing* the game is over. If an opponent makes a crossing, then play will continue. Crossed pieces can no longer move and a player cannot capture them. The game continues until one player has more *crossed* pieces than the other. Games can end in a draw. Draws occur when both players complete *crossings* and have an equal amount of pieces on each home row with no further moves available. In Figure 2.7 it is Black's move. Black can cross with [A7,A6] - [A8], then White will

respond with a second crossing [H2-G1]. Black then responds [A7-B8] and the game is a draw since neither player has any legal moves left.

There are instances where no legal moves left for a player is not indicative of a draw. A player may lose all of their pieces, or have a crossed piece and then lose the rest of their pieces during game play. Both instances result in a situation where one player may have no legal moves left. When a player loses all their pieces, they lose the game. If a player has no legal moves left, and the opposing player can eventually make an additional crossing, then the opposing player wins. In Figure 2.8 it is White's move. After capturing Black's piece on A7, Black will have no legal moves left and White is the winner.
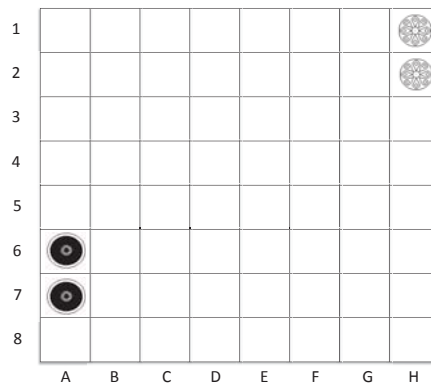
Figure 2.7: Black to Move – Game is a draw.

2.11.5   Basic Strategies.

2.11.5.1   Softening.

If a large phalanx is about to capture another, often times it is best to move the leading piece towards the threatening phalanx. The opponent must first capture the singleton piece before trying to capture the original phalanx. This can lead the attacker into situations
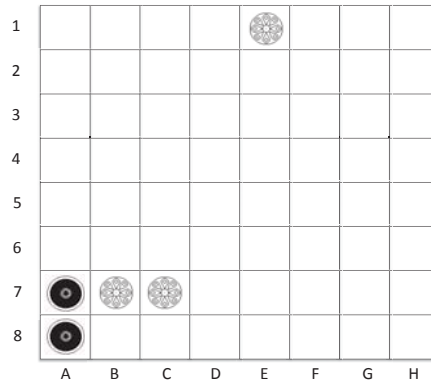
Figure 2.8: White to Move – After capturing A7, White wins.

where the defender can bring other phalanxes into play and recapture the attacking phalanx or use the cutting strategy.

2.11.5.2 Cutting.

The size of a phalanx dictates its mobility and capturing power. One way to mitigate both is to split the opponents phalanx in some manner. Cutting a phalanx reduces its offensive and defensive capabilities and is useful in slowing down an opponent's ability to launch deep attacks into one's territory.

2.11.5.3 Channels.

Carving channels into enemy positions is a valuable strategy, especially in row 2 for White and row 7 for Black. This allows for crossings where the enemy either cannot *cut* the long phalanx, thus allowing further immediate crossings, or opens up long channels of free movement for other phalanxes to exploit.

2.11.5.4 Close Gaps.

Connected pieces are stronger than singleton pieces. As phalanxes move from their original positions, they often leave holes in the home rows. Closing these gaps avoids channels and builds larger defensive phalanxes that are vitally important for thwarting

29

enemy crossings. This is a defensive maneuver for both players. Since one cannot capture an enemy piece that has crossed, it is best to empty the back row, focusing on defending rows 2 and 7 respectively. However, in certain situations, see *Blocking*, having pieces on the home row can be advantageous.

2.11.5.5 Blocking.

As the game progresses, more enemy pieces come closer to making crossings. In these situations, a player may move their pieces into gaps on the back row to prevent singleton phalanxes from crossing over. This maneuver is quite effective when used in conjunction with sweeping.

2.11.5.6 Sweeping.

In this strategy, a player builds a mobile horizontal phalanx of 3 to 4 pieces on row 2 or 7 (depending on the player's perspective). This *sweeper* is used to capture any pieces landing on those rows, thus preventing crossings from occurring. A sweeper unit can also use the tactic of softening by throwing itself into the way of an incoming enemy phalanx to prevent immediate crossings.

2.12 Epaminondas

The following sections outline the rules and basic strategies for *Epaminondas* [22].

2.12.1 Overview.

Epaminondas is played on a 14 x 12 checkered board with 28 black and 28 white pieces. The original board is set up in the starting position seen in Figure 2.9. White plays first, followed by black, and so forth. Players cannot pass. Both players' goal is to move pieces, called phalanxes, to their opponent's back row. Whoever has more pieces on their opponent's back row after one full turn, wins the game.
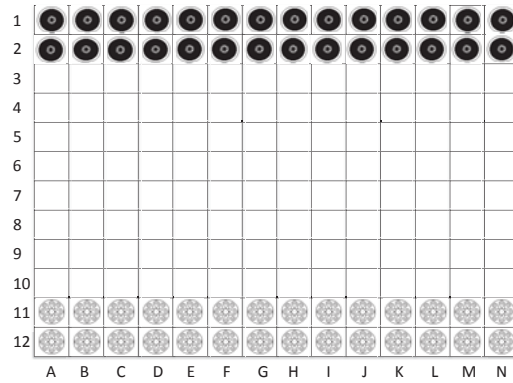
Figure 2.9: Epaminondas Starting Position.
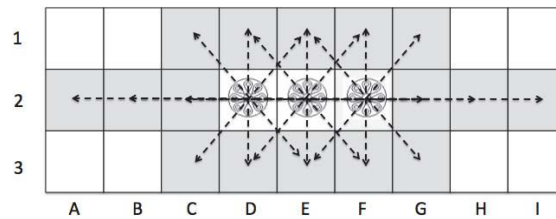
Figure 2.10: Example Phalanx Moves in Gray.

2.12.2 Phalanxes and Movement.

A phalanx is a connected group of one or more pieces. Theses pieces must be horizontally, vertically or diagonally inline with one another. According to Abbott, these phalanxes are representative of Ancient Greek battle formations where hoplites lined up side by side, and front to back in squares, to face off against enemy armies [2]. Phalanxes of size one, can move in any direction. They cannot move onto occupied squares. Groups of two or more phalanxes can only move in straight, orthogonal or diagonal lines (forward and backward) depending on the orientation of the phalanx. Pieces can belong to multiple phalanxes at once. The number of spaces a phalanx can move is less than or equal to the

31

number of pieces in the phalanx. A one piece phalanx can move one space, a two piece phalanx can move one or two spaces, a three piece phalanx can move one, two, or three spaces and so on. Phalanxes can split for moves as well. For example, a player can split a larger phalanx into a smaller one and move the appropriate spaces accordingly. Phalanxes cannot move through friendly or opposing pieces. Only in the case of a legal capture can a phalanx move into an occupied square. Figure 2.10 provides an example of the number of moves available to a player in one small area. The list of possible moves is as follows:

- Phalanxes of Size One:

    - D2-C1, D2-C2, D2-C3, D2-D1, D2-D3, D2-E1, D2-E3, E2-D1, E2-E1, E2-F1, E2-D3, E2-E3,E2-F3, F2-E1, F2-E3, F2-F1, F2-F3, F2-G1, F2-G2, F2-G3

- Phalanxes of Size Two:

    - [D2,E2]-C2, [D2-E2]-B2, [F2,E2]-G2, [F2,E2]-H2

- Phalanxes of Size Three:

    - [D2,E2,F2]-C2, [D2,E2,F2]-B2, [D2,E2,F2]-A2, [F2,E2,D2]-G2, [F2,E2,D2]-H2, [F2,E2,D2]-I2

This simple position contains 30 possible moves, demonstrating the complexity of *Epaminondas* positions.

2.12.3 Capture.

In order to move onto an enemy occupied square, the number of pieces in the attacking phalanx must outnumber the number of pieces in the defending phalanx. If the attacking phalanx is of equal size or smaller, then movement stops at the square in front of the occupied enemy square.

If a capture occurs, the lead piece of the attacking phalanx occupies the square where the lead piece of the defending phalanx resided. A player loses their entire defending phalanx when a capture occurs. Figure 2.11 shows an example of a legal capture.
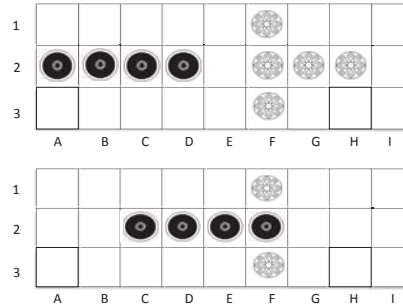
Figure 2.11: After Capture: [E2,D2,C2,B2] x [F2,G2,H2].

If a White piece resided on I2, then White would have avoided capture.

2.12.4 Objective.

The objective of the game is to move one's pieces across the board to the opponent's back rank. If, at the start of White's turn, White has more pieces on Black's back row than Black has on White's back row, White wins. The same applies at the start of Black's turn. The following descriptions of Figure 2.12 and Figure 2.13 clarify winning and continuing game conditions.

In Figure 2.12, White moves onto Black's back row [H2,H3,H4]-[H1]. Black has two options, immediately capture the piece, or move a piece onto White's back row. In this situation, Black can do neither. Black will make a move, and then, because it is White's turn, and White has more pieces on Black's back row than Black has on White's back row, White wins the game.
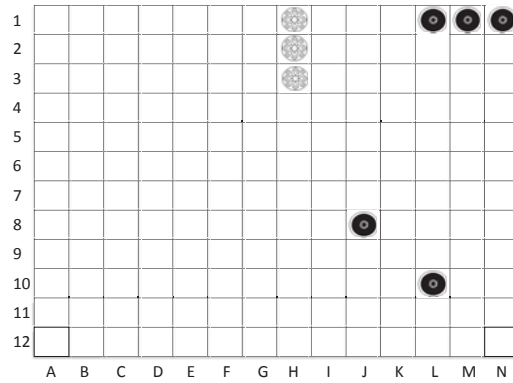
Figure 2.12: White *crosses*. [H2,H3,H4] - [H1].

In Figure 2.13, Black can respond by capturing White's piece [L1,M1,N1]x[I1]. Black can also move onto White's back row [L10,K9,J8]-[N12]. Either move results in an equal number of pieces on each opposing back row; zero in the former, one apiece for the latter. After Black moves, the game would continue.
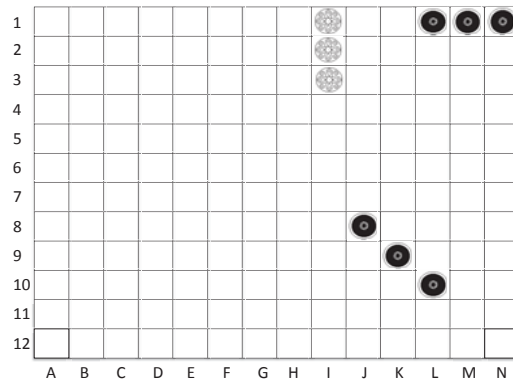
Figure 2.13: Black to Answer.

Pieces moved onto a back row are available for future moves but, often times, once they are on an opponent's back row they stay until captured or the game ends. One

additional win condition, not covered in the article by Handscomb, is exhaustion of pieces. Exhaustion of opposing pieces is a *de facto* win condition for a player. Finally, to help alleviate draws, Abbott added a rule of symmetry [22]. A player cannot move their piece onto the row furthest from them if it creates a pattern of left-to-right symmetry.

2.12.5 Puzzles.

The following puzzles were first published in the Handscom article on *Epaminondas* [22]. Abbott personally authored these puzzles and they serve as the only known test cases for *Epaminondas*. These puzzles exemplify the complexity of the game. The implemented *Epaminondas* agent solved these four puzzles with the Min-Max algorithm. The solutions follow the puzzle descriptions.
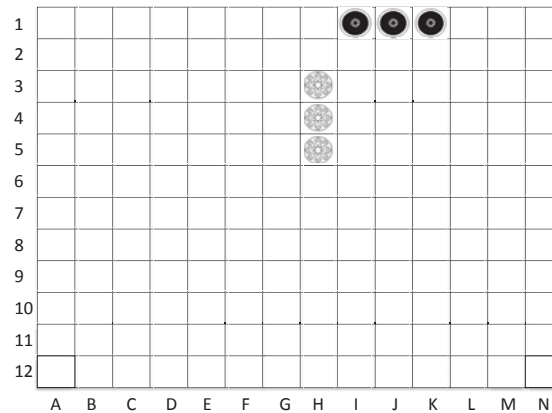
Figure 2.14: Puzzle 1: White to win in three.

Figure 2.14: It is White's turn and can win in three full turns (White move + Black move = 1 full turn). The main threat is dropping the three-piece phalanx onto Blacks back row, however, doing this move first allows Black to recapture easily. Solution: White: [H3,
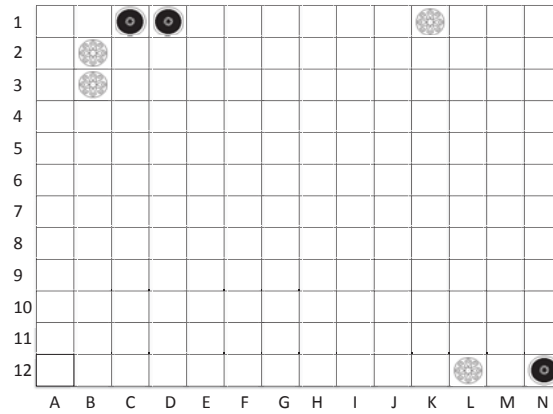
Figure 2.15: Puzzle 2: White to win in two.
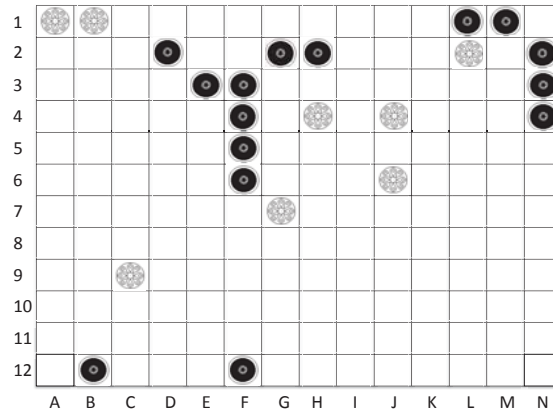
Figure 2.16: Puzzle 3: White to win in four.

H4, H5]-H2. Black: [I1, J1, K1]-J1. White: H2-G1. Black: [I1, J1, K1] x G1. White: [H3, H4] x H1. Black: N/A. White wins since Black cannot recapture.

Figure 2.15: Again, White threatens another crossing onto Black's back row and again, Black has enough defenders to prevent this. White must force Black to move its two-piece phalanx on Row 1 in order to win. White wins in two full turns. Solution: White: [L12]-

36

M12. This forces Black to move its two-piece phalanx. All possible positions are a loss. For example, [C1, D1]-E1, White responds with B2-A1 and Black is too far away to capture. Moving towards A1 results in a White capture and win.

Figure 2.16: In this example, White has fewer pieces than Black, however, White is threatening to make a crossing. The correct move for White is subtle but forces a win. White wins in four turns. Solution: White: J4-I3. Black: N2-N1. White: L2-K2. Black: [N1, M1, L1]-I1. White: K2-L1. Black: [N1, M1, L1] x L1. White: [I3, J4] x L1. Black: N/A. White wins since Black cannot respond by capturing White's piece or by making a crossing.

2.12.6   Basic Strategies.

2.12.6.1   Softening.

If a large phalanx is about to capture another, often times it is best to move the leading piece towards the threatening phalanx. The opponent must first capture the singleton piece before trying to capture the original phalanx. This can lead the attacker into situations where the defender can bring other phalanxes into play and recapture the attacking phalanx or use the cutting strategy.

2.12.6.2   Cutting.

Often times it is best to cut opposing phalanxes in two. This reduces their mobility and attack capability. Instead of a 5-piece phalanx, one can reduce it to two, two-piece phalanxes that are more vulnerable.

2.12.6.3   Channels.

Carving channels into enemy positions is a valuable strategy, especially in an opponent's back row. This allows for crossings where the enemy either cannot recapture completely (defending phalanxes are now smaller) or cannot recapture at all (reduced to one piece phalanxes).

2.12.6.4 Close Gaps.

Connected pieces are stronger than singleton pieces. As phalanxes move from their original positions, they often leave holes in the home rows. Closing these gaps avoids channels and builds larger back row phalanxes that are vitally important for thwarting enemy crossings.

2.12.6.5 Sweepers.

As with *Crossings*, sweeper phalanxes are effective at limiting an opponent from making crossings. Unlike *Crossings*, however, it is best to place long sweepers on the home row since *crossed* pieces can be captured. Often times, these sweepers can turn the tide of the game as crossed pieces are removed, thus increasing the likelihood that one ends up with more *crossed* pieces on the enemy home row.

2.12.6.6 Piece Domination.

In many capture games having more pieces than an opponent is an indicator of a favorable board position. For *Epaminondas*, the more pieces a player has the more likely they can traverse the board and make a *crossing*. In addition, piece dominance is indicative of offensive and defensive potential.

2.13 Summary

This chapter reviewed the current literature on gaming research in AI. It proposed that game research is important because it brings solutions to real world problems such as combating urban traffic and defining trust in medical care as well as testing algorithmic limits. Additionally, it outlined the major algorithms in use today and domains where they currently fail. This chapter established the idea that heuristic values may increase the UCT algorithm's game playing ability and that adding heuristic information in UCT is rather straightforward, avoiding the complexity associated with RAVE and Winands' $\alpha\beta$ play outs. Finally, this chapter introduced the rules and strategies for *Crossings* and *Epaminondas*.

III. Methodology

This chapter describes the approach used to answer the research questions presented in Chapter 1. First, it restates the research goals. Section 3.2 follows with an overview of Min-Max $\alpha\beta$ agent development and defines the six major heuristics used to evaluate the board state. Next, Section 3.3 describes the common approach used to derive the state-space and game-tree complexities of games. Section 3.4 defines the two Monte-Carlo (MC) based algorithms, Upper Confidence Bounds applied to Trees (UCT) and Heuristic Guided UCT (HUCT), used in the experiments. Section 3.5 describes the testing environment. Finally, Section 3.6 details the performance metrics used to evaluate the implemented MC algorithms.

3.1 Research Goals

The goals of this research are multifold. One, construct game playing agents for *Crossings* and *Epaminondas* to establish game-tree complexity estimations for each game and determine their solvability. These agents then become the primary opponent for the Monte-Carlo (MC) based search agents. Second, modify UCT to include limited heuristic knowledge, then assess the performance of UCT and its modification, HUCT, in a game of low complexity (*Crossings*) and in one of high complexity (*Epaminondas*). The performance of these algorithms is compared to reach conclusions about the effectiveness of each algorithm in both environments when compared to an $\alpha\beta$ agent, as well as each other. Finally, analyzing their performance can determine if either domain is dominated by Min-Max with Alpha-Beta ($\alpha\beta$) pruning.

3.2 Agent Development

Since no known agents for *Crossings* and *Epaminondas* exist, the first step towards developing the game-tree complexity was building agents to play the games. The Min-Max

$\alpha\beta$ algorithm was selected to represent the baseline agent for all experiments. Building the Min-Max $\alpha\beta$ agent consisted of encoding heuristics from similar games such as *Lines of Action*, *Chess* and *Go*. Human game play experience also guided heuristic generation. Heuristic development stopped when a Min-Max $\alpha\beta$ agent, set to a search depth of 3, returned a move within 15 seconds and played at a novice level. Being able to defeat a human opponent playing at a beginner level defines a novice level of play.

The method for developing a novice level of play consisted of implementing a basic Min-Max $\alpha\beta$ agent and then adding, testing, and refining heuristic functions, qualitatively measuring their effects on game play. The first versions of the *Crossings* and *Epaminondas* agents utilized basic alpha-beta search defined by Knuth [29]. Knuth's Min-Max $\alpha\beta$ algorithm was used for multiple reasons. One, it is well documented and researched. Two, it is easy to implement. Finally, any basic evaluation function immediately begins pruning the search space. This pruning allows the agent to search more nodes in less time, enabling better play. For the game-tree complexity experiments, Min-Max $\alpha\beta$ versus Min-Max $\alpha\beta$ appeared more likely to produce a better estimation of the game tree space over random only players. Additional refinements improved the performance of the agent.

The first major improvement was the incorporation of transposition tables. The agent uses a Zobrist hash to save a visited board state into an array [62]. Before the agent evaluates a board state, it consults the transposition table. If a "hit" occurs, then the agent receives the value of the board state, making reevaluation unnecessary. Most games, when transposed to graphs, contain cycles. During a regular Min-Max $\alpha\beta$ search this means an agent may reevaluate the same board state multiple times, thus slowing its search and affecting its responsiveness. The more nodes the agent can search in the allotted timeframe, the better it will play. Transposition tables resulted in considerable speed up for *Epaminondas*. In early testing, they enabled the agent to almost double the amount of nodes processed per second.

Next, leveraging heuristics from *Lines of Action* and the ancient Egyptian game *Seega*, the evaluation function was modified to take into account bad zones as well as piece counts. The outermost columns of *Crossings* (Columns A and H) and *Epaminondas* (Columns A and N) appear relatively weak for phalanx formations since phalanxes along these columns are highly vulnerable to horizontal and diagonal attacks. In addition, early versions of *Epaminondas* agents often left singleton phalanxes stranded on these columns versus bringing them into larger phalanxes. Adding the bad zone heuristic resolved this problem. For *Seega*, researchers used piece counts (number of ones pieces subtracted from the number of opposing pieces) to help guide their agents [6]. This heuristic also assisted the agents performance.

Seven heuristics for the game *Lines of Action* include threats, solid formations, mobility, blocking, centralization, material advantage, position values, and initiative [52]. As game knowledge of *Crossings* and *Epaminondas* grew through human experience and agent self-play, six further strategies evolved: softening, cutting, channels, close gaps, blocking, and sweeping. The following sub-sections describe the major heuristic functions used by the Min-Max $\alpha\beta$ agent to encode these strategies.

3.2.1 Mobility.

In both games, mobility is a key to victory. The agent uses two functions to calculate the mobility score. First, it sums the total number of squares that all the phalanxes can traverse and then divides that by the number of moves available. For example, if a player has only one piece left on the board and it can move freely in all eight directions, the state is scored as $\frac{1}{8} = .125$. The second function merely tracks the largest distance that can be covered by any of the player's phalanxes. In the previous example, the value is 1. The mobility score for this player is 1.125.

3.2.2 Material Dominance.

In many games, possessing more pieces is indicative of a winning position. This function returns the difference of the sums of opposing pieces. A negative value indicates material advantage for the opponent while a positive value indicates otherwise. A value of zero means neither player is ahead as far as material is concerned.

3.2.3 Crossing.

The object of the game is to cross to an opponent's back row. This heuristic captures the idea of a crossing by summing the number of pieces on the opponent's back row in the current board state. One point is assigned for each piece. If a player has two pieces on the opponent's back row then the position is assigned two points, three pieces equals three and so on.

3.2.4 Center of Mass.

In many board games such as *Lines of Action* and *Chess*, the center squares play an enabling role for winning. This is well explored in *Chess* with many openings concentrating on either controlling, or contesting, the middle of the board. This function calculates the Euclidean distance for all pieces from the center of the board for each side. The agent subtracts these scores. A positive value indicates a higher center of mass for a player.

3.2.5 Home Row Defense.

As crossings are important to winning, preventing one's opponent from making crossings is vital. One defensive maneuver in both games is to build a large phalanx on the back row (*Epaminondas*) or the next to last row (*Crossings*); refer to *Sweeping* in Chapter 2. The agent uses these phalanxes to capture crossed pieces in *Epaminondas* or prevent crossings in *Crossings* by moving in the path of threatening phalanxes. The function created to encode this idea calculates the largest contiguous phalanx on the home row in *Epaminondas* or next to home row in *Crossings*. For example, if a player has four pieces connected on their home row, then this function returns a value of four.

3.2.6 Territory.

Owning territory is important in board games such as *Chess* and *Go*, with the former being its winning condition. The idea of territory in *Crossings* and *Epaminondas* is a little more abstract since territory can be contested, pieces blocking one another or instances where one can capture the other. In a manner similar to *Go*, this function looks at each piece individually and looks at each surrounding space around it (all eight directions). If a space is empty, then the value of that particular square is given +1. If the space is occupied by a friendly piece, the square is given a +1. Otherwise, the square is given +0. One can see that if a piece is in the middle of the board, with no enemy pieces around it, its territory score will be an 8. The total territory score is the summation of all the squares surrounding the player's pieces not occupied by enemy pieces. The opposing territory score is also calculated and then subtracted from the original player's score. A positive value indicates a strong territory score. In the future, this function should take into account contested squares, *i.e.* squares that are contested by both sides, as well as weighing certain board squares more heavily than others.

3.3 State-Space and Game-Tree Complexity Analysis

The state-space complexity of a game is defined as the number of legal positions reachable from the initial board position [3]. One method for estimating the state-space for a game is proposed by Allis in his work *Searching for Solutions* [3]. Allis calculates the values possible for each space on the board. By assigning each space one of three possible values: white, black, or null, a loose estimate of the state-space for *Crossings* is on the order of 10^{30}, while *Epaminondas* is close to 10^{80}. Winands uses a stricter mathematical approach to tighten the estimate for the state-space of a game in his thesis on *Lines of Action* [52]. Winands bases his formula on Schaeffer and Lake's work on *Checkers* [51].

Winands' State-Space Complexity Equation (3.1):

$$\sum_{B=1}^{maxBPieces} \sum_{W=1}^{maxWPieces} \binom{numSquares}{B} \binom{numSquares - B}{W} \tag{3.1}$$

where *B* equals the number of black pieces and *W* equals the number of white pieces
[51]. Winands further refines his state-space estimates by eliminating positions that, while
theoretically possible, are unachievable through play [51]. These are called spurious states
[61]. The only states removed for *Crossings* and *Epaminondas* were those where each side
possessed one piece left on the board. These positions are impossible to reach through legal
game play. However, unlike *Lines of Action*, both *Crossings* and *Epaminondas* can have
positions where a side has two pieces left, while the other has zero. This situation is an
automatic win for that player and is the result of legal moves (captures).

Although deriving the state-space is rather straightforward, game-tree complexity
analysis is a little more complicated. The game-tree complexity of a game is defined as
"the number of leaf nodes in the solution tree of the initial position of the game," where
the solution tree for a move is of full width and is of sufficient depth to determine the
game-theoretic value of that move [3]. One can view the game-tree complexity of a game
as an estimate of the game's decision complexity. If the game is small enough, one can
enumerate all possible moves from all possible positions. However, in all but trivial games
such as *Tic-Tac-Toe*, this is infeasible. One must build an agent to play multiple games to
find the average length of a game as well as the average branching factor per move. In other
words, how many turns does a normal game contain, and how many moves are available to
a player per turn.

For each game, one thousand self-play games established the baseline to determine
the game-tree complexity. From these self-play games, average game lengths and the
average branching factors were determined. Equation 3.2 presents the formula for deriving
an estimate of the game-tree complexity.

Estimate of Game-Tree Complexity Equation (3.2):

$$BranchingFactor^{(GameLength)}$$ (3.2)

This estimate relies heavily on the correctness of the heuristic value embedded in the Min-Max $\alpha\beta$ agent. A poor heuristic may result in an over or under estimation of the game length and branching factor. The agent used Min-Max $\alpha\beta$ search with all four Min-Max $\alpha\beta$ enhancements: move ordering, killer moves, history heuristic, and transposition tables during play. To enable fair play, and produce tighter results, the agent randomized the first three moves for each player. This is similar to Winands' [52] and Schadd's [35] initial research efforts where they biased the Min-Max $\alpha\beta$ algorithm to produce "real" game play for their respective games . The results of the thousand self play matches give a good estimation of the game-tree spaces for *Crossings* and *Epaminondas*.

3.4 Monte Carlo Methods

Monte-Carlo (MC) methods are the focus of many researchers today, especially for the game of *Go* (refer to Chapter 2, section 2.8 for further details on MC search algorithm evolution). MC based algorithms are notoriously noisy where results in play can vary widely from one game to the next. This is due to the stochastic nature of MC methods. Both UCT and HUCT played against a tuned Min-Max $\alpha\beta$ agent set to a depth of 3. The random factor associated with Min-Max $\alpha\beta$ self-play was removed. Each MC based algorithm played 5,000 games as White and 5,000 games as Black to identify advantages for either color, if they existed at all. Finally, decision times were set to 1, 5, 10, and 30 seconds. The agent simulated 10,000 games per time interval during testing, lending support to the data observed.

To further mitigate interference with the MC agent, each game was launched as a separate thread and only five threads were run at a time to avoid overloading machine processors. This ensured each thread received approximately the same amount of

45

processing time in the allotted time interval. Since MC methods were limited to a time window, placing too much strain on the processors would result in fewer simulations per second negatively impacting the MC agents performance. Keeping the core utilization threshold to 80 percent produced equivalent results in preliminary testing across all three machines.

The UCT algorithm provided the baseline MC agent for MC assessment. UCT node selection was guided by:

UCT State EvaluationEquation (3.3):

$$Value = X_j + C * \sqrt{\frac{\ln(n)}{n_j}} \tag{3.3}$$

The C constant value varies with each domain. After initial testing, 0.445 provided a balance between exploration and exploitation using UCT. The algorithm performed the common random play out for each simulated game, backpropagating 1 for a win, -1 for a loss, and 0 for a draw once complete. The modified UCT algorithm included the heuristic value of the node in the following manner:

Heuristic Guided UCT Node EvaluationEquation (3.4):

$$Value = X_j + HValue(State) + C * \sqrt{\frac{\ln(n)}{n_j}} \tag{3.4}$$

After preliminary testing, C was set to 0.667 allowing for fuller exploration of each level. The HValue(State) term represents the call to the heuristic function used by the Min-Max $\alpha\beta$ agent. This call costs computational time as the agent has to make the move, evaluate it, and then revert the game state. The goal was to guide the agent towards more promising parts of the tree through the heuristic value to overcome the loss of simulations performed. It followed the same play out and backpropagation scheme of the normal UCT agent. The agent only calculated the heuristic value of a node at the expansion step avoiding recalculations if the agent selected the node for play out later in its time interval.

3.5 Environment

Both *Crossings* and *Epaminondas* provide the environment for all the experiments run during testing. *Crossings* establishes a baseline from which comparisons can be drawn. The similarities between the games allows for algorithmic comparison as they cross from a lower complexity to a higher one. Data collected also grants insight into the games themselves. For example, Min-Max $\alpha\beta$ agents show that White appears to hold a slight advantage in *Crossings* and in *Epaminondas*.

All heuristic refinement, and complexity experiments were ran on a 2.9 GHz Intel Core i7, 8 GB 1600 MHz DDR3, Mac Book Pro running Mac OS X Lion 10.7.5 using Eclipse Version: Juno Service Release 1 Build id: 20120920-0800. Monte-Carlo agent experiments were run on three 3.1 GHz Intel Xeon Dells, running Windows 7 Enterprise Edition 6.1 using Eclipse Version: Juno Service Release 1. The native operating systems scheduled game simulations without interference or modification by the programs running the agents.

3.6 Performance Metrics

An algorithm's win ratio is the primary measure of success. Game length and simulations achieved per turn were compared to win ratios to gain additional information about the effectiveness of the MC search algorithm in question as well as the agent's behavior in the underlying testing environment.

MC Based Algorithm EvaluationEquation (3.5):

$$WinRate = \frac{numWins}{numGames} \qquad (3.5)$$

3.7 Summary

This chapter introduced the approach taken to answer each research question. It laid the groundwork for the experiments and data results chapters that follow. The chapter identified how basic agents for each game were constructed. Furthermore, it defined the

heuristics used to refine their searches. Additionally, this chapter presented MC agent testing and the performance metrics used to assess an their performance.

IV. Experiments and Model Design

This chapter outlines the three experiments implemented to answer the research questions. Section 4.1 details the construction of the novice game-playing agent and defines an additional eight heuristics used by the Min-Max $\alpha\beta$ agent to evaluate the board state. Furthermore, it outlines the parameters for establishing novice play. Section 4.2 outlines how the agent derived the average game lengths and branching factors to calculate the game-tree complexity for *Crossings* and *Epaminondas*. Finally, Section 4.3 describes the testing of the MC based agents.

4.1 Experiment One: Agent Development

The first experiment consisted of a series of human versus agent games designed to create a novice level Min-Max with Alpha-Beta ($\alpha\beta$) pruning agent for each domain. After encoding the rules outlined in Chapter 2 into a basic Min-Max $\alpha\beta$ algorithm with move ordering, killer moves, the history heuristic, and transposition tables, the experiment became focused on heuristic refinement. In addition to the heuristic functions outlined in Chapter 3, the following heuristics were added to the agent's state evaluation:

- Bad Zones: number of one's pieces on outside columns minus opponent's pieces in outside columns

- Average Phalanx Size: reward equals the average phalanx size in current position

- Largest Phalanx Bonus: equals the largest phalanx one owns

- Average Distance: average distance an agent can cover

- Longest distance: greatest distance that can be traversed unimpeded

- Greatest Capture: size of the largest enemy phalanx that can be captured

49

- Pieces Available for Capture: sum of all opposing pieces one could capture

- Average Capture: average number of pieces that can be captured

The threshold for move return was set to 15-seconds for a Min-Max $\alpha\beta$ agent set to a search depth of 3. Once an agent met this threshold, and played at a novice level against a human player, agent development stopped. The definition of novice play is a qualitative one. No known agents play *Crossings* or *Epaminondas*. The determination of novice level of play was based upon the agent playing *good* moves and winning against beginner level strategies.

4.2   Experiment Two: Complexity Development

Chapter 3 defines the formula used for state-space calculation. For the game-tree complexity, the agent ran 1,000 self-play games. For both domains, the agent was given 30 seconds to conduct a move. In *Crossings*, the search depth was set to 5 since the novice agent could return a move within a 30 second timeframe. The *Epaminondas* agent was set to 3 since it could not return a depth of 5 search in under 30 seconds. In order to produce different games, the agent introduced a random value set to 0.5 for the opening move. As the game progressed, the probability of a random move diminished by 0.5 after each players move to a set random probability of 0.01 after a few moves. This ensured the Min-Max $\alpha\beta$ agents played different games each time. Otherwise, Min-Max $\alpha\beta$ agents would play the same game continuously providing little to no knowledge about game characteristics. The agent sent all board states and the number of moves available per turn to a text file for later analysis.

4.3   Experiment Three: Assessment of Monte-Carlo Based Agents

Due to the stochastic nature of MC based search agents, a high number of simulations were run to gain confidence in the results. Each algorithm played 10,000 games per time interval. For example, UCT played 10,000 games against a Min-Max $\alpha\beta$ agent at 1 second,

then another 10,000 games at 5 seconds, and so on. The agents played 5,000 times as White and 5,000 times as Black. This avoided a biased data set where one side may have dominance over the other and thus, skew the results. Again, *Crossings* and *Epaminondas* are untested domains so these tests also provide information about one player's advantage over the other. Agents played 10,000 game sets at 1, 5, 10, and 15-second time intervals to assess MC performance as time increased across both domains. The agent wrote all game states, number of simulations completed per turn and win-loss records to a text file for later analysis.

4.4   Summary

This chapter explained the development of Min-Max $\alpha\beta$ agents to play both *Crossings* and *Epaminondas* delving into heuristic evaluation functions and how they apply to the overall heuristic evaluation of a board state. This data enables an estimate of the game-tree complexity for each domain. Furthermore, this chapter reviewed how the MC agents were assessed. The first MC agents, UCT, is well known and heavily used in AI research today. The second, HUCT, is a modification of the UCT algorithm's node expansion and selection stages, along the lines of heuristic guided search proposed by Winands in his work on *Lines of Action*. The basic premise is to incorporate heuristic game knowledge to guide the MC agent to *better* parts of the search tree early, hoping to avoid *poor* areas of the tree, improving UCT's performance. A more detailed explanation of both algorithms resides in Chapters 2 and 3.

V.   Results and Data Analysis

This chapter presents the results of the experiments detailed in Chapter 4. It begins with the development of a novice Min-Max $\alpha\beta$ agent to play both games. Sections 5.2 and 5.3 provide the results of state-space and game-tree complexity computations as well as general observations about each game. This is followed by a comparison of the game domains. Section 5.5 provides the results of Monte-Carlo (MC) based agent play. Here, an assessment of their performance is quantitatively compared to the baseline Min-Max $\alpha\beta$ agent as well as each other. Finally, section 5.6 outlines general observations drawn from the MC agents' performance compared across both domains.

5.1   Game Playing Agents

The agents developed through the methods and heuristics outlined in Chapters 3 and 4, eventually achieved a novice level of play in the *Crossings* domain. As stated, novice level play equates to winning against a human beginner player. While qualitative in nature, no known agents exist to play *Crossings* to enable quantitative results. The main guidelines for improvement are the responsiveness of the agent as well as quality of its move selection. After a series of games, the *Crossings* agent, set to a search depth of 5, returned novice level moves in under 15 seconds. *Epaminondas* proved more difficult.

The *Epaminondas* agent achieved a beginner level of play. Eventually, through heuristic refinement, the agent, set to a search depth of 3, returned beginner to novice level moves within 15 seconds. The agent plays aggressively but the depth limit precludes large phalanx build up that is vital to better play. A player can take advantage of the agent's aggressive nature and quickly develop strategies to beat it. Future work needs to refine the heuristics to prune away more of the search space to increase the agent's performance.

Setting a goal of 30 seconds for a depth of 5 search is not unreasonable for such a complex game.

5.2 Properties of Crossings

5.2.1 State-Space Complexity.

Using Winands' formula described in Chapter 3, the state-space complexity for *Crossings* is 3.63×10^{27}, placing it above *Lines of Action*, *Fanarona*, and *Checkers* [35, 43, 51]. Winands' method reduced the Allis based state-space estimate by 10^{11} positions. A complete listing of the number of possible moves per pieces left on the board is in the appendix (Table A.1).

5.2.2 Game-Tree Complexity.

In order to derive the game-tree complexity the agent played 1,000 self play games using the Min-Max $\alpha\beta$ algorithm set to a depth of 5. This data enabled the calculation of the average length of a game as well as the average branching factor. The average game length for *Crossings* is 39 with a standard deviation of 31. The average branching factor is 110 with a standard deviation of 27. The formula for estimating the game-tree complexity of a game is raising the branching factor by the game length. This yields a game-tree space of 10^{79}, placing *Crossings* above *Fanarona*, *Othello* and *Lines of Action* [3, 35, 51]. It is well below *Chess* and *Go*. However, surpassing *Othello* and *Lines of Action* is interesting. It highlights the fact that the complexity of movement, in this case allowing multiple pieces to move at once, directly impacts the overall complexity of the game by expanding its branching factor. For *Crossings*, although played on the same size board as *Lines of Action*, move complexity increased the game complexity by 10^{10} states. Taking into account *Crossings'* high state-space and high game-tree complexities, *Crossings* is unsolvable by current methods.

Figure 5.1 and Figure 5.2 show the distribution of the average games lengths and branching factors for *Crossings*. Game length equals 1 turn (*i.e.* 1 turn = White's move or

Black's move, sometimes referred to as *ply*) The diamond in the upper box plot for both figures represents the mean whose width is a 95% confidence interval of the mean.

Summary Statistics

Mean   39.748
Std Dev   31.920517
Std Err Mean   1.0094154
Upper 95% Mean   41.728818
Lower 95% Mean   37.767182
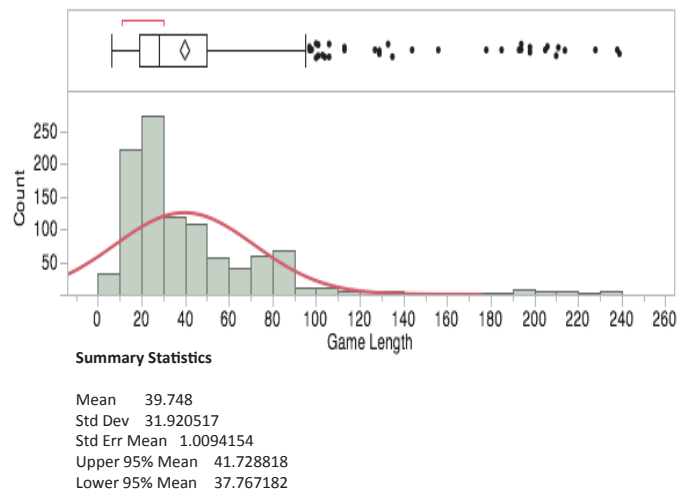
Figure 5.1: Crossings Game Lengths.

Summary Statistics:

Mean   110.788
Std Dev   27.814037
Std Err Mean   0.8795571
Upper 95% Mean   112.51399
Lower 95% Mean   109.06201
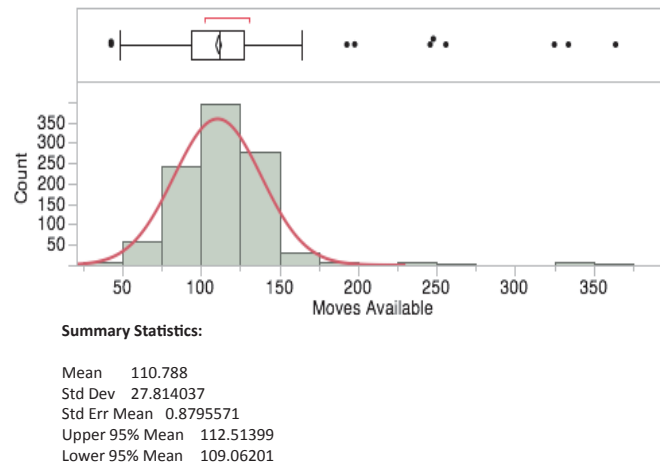
Figure 5.2: Crossings Branching Factor.

5.2.3   Game Observations.

Figure 5.3 shows that the number of moves increases quickly in the first few turns. After turn 10, the average number of moves available to a player drops rapidly as each player captures and loses pieces. This indicates that players come into conflict quickly in *Crossings* leading to a tactical opening sequence. An analysis of 1,000 self-play games of a Min-Max $\alpha\beta$ agent set to a depth of 5, with a time threshold of 30 seconds per move, showed little advantage for either side. For all trials, the Min-Max $\alpha\beta$ agent played stronger as White when playing against Upper Confidence Bounds applied to Trees (UCT) and Heuristic Guided UCT (HUCT) agents.

Mean: 32.32
Standard Deviation: 18.96
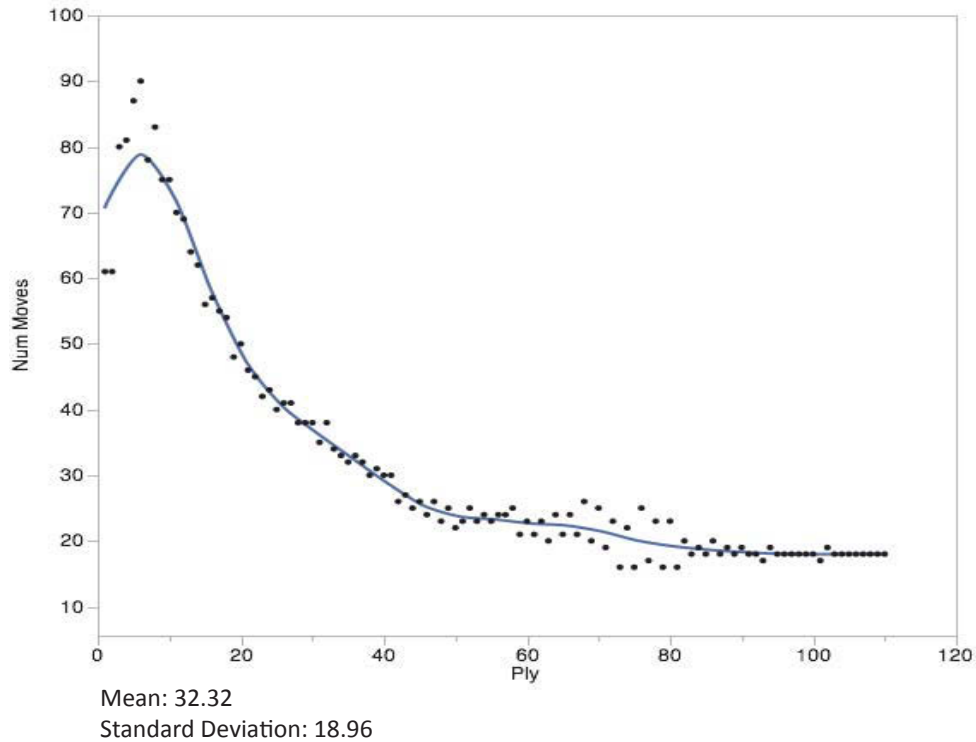
Figure 5.3: Crossings Branching Factor Over Time.

56

5.3 Properties of Epaminondas

5.3.1 State-Space Complexity.

Using Winands' formula, the state-space complexity for *Epaminondas* is 2.41×10^{61}, placing it above *Checkers*, *Lines of Action*, and *Chess* [3, 35, 43, 51]. Winands' method reduced the state-space estimation by 10^{19} positions. A complete listing of the number of possible moves per pieces left on the board is in the appendix (Table A.2).

5.3.2 Game-Tree Complexity.

Using the same method applied to *Crossings*, the average game length for *Epaminondas* is 56 with an average branching factor of 283. These results yield a game-tree space of approximately 10^{137}. This places *Epaminondas* above *Chess* (10^{123}) [37] and below *Go* 10^{360} [3]. It also places *Epaminondas* squarely in the category of *unsolvable* by current methods according to Herik's defined categories [24]. Figure 5.4 and Figure 5.5 show the average games length and branching factor for *Epaminondas* (Game length = 1 turn = 1 ply). Again box plot diamonds represent the mean with widths showing the 95% confidence interval of the mean.
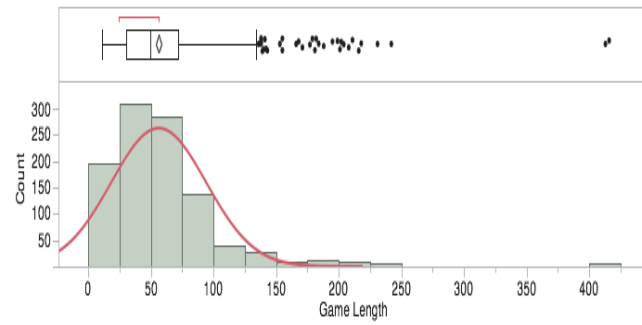
Figure 5.4: Epaminondas Game Lengths.
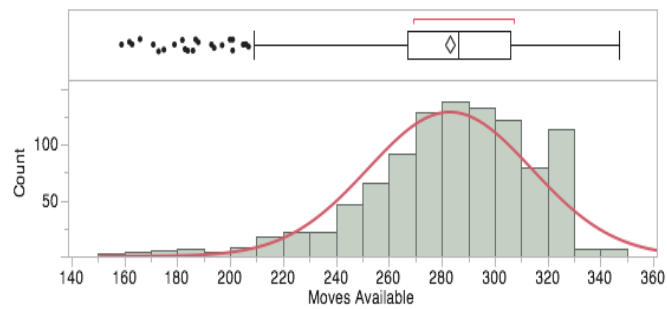
Figure 5.5: Epaminondas Branching Factor.

5.3.3   Game Observations.

Figure 5.6 shows that the number of moves available to a player rises quickly as the player builds phalanxes up to ply 20. This slowly diminishes as each player captures pieces. The distance between players allows for greater phalanx development for each player. Instead of a player being able to reach the strategic goal (a crossing) early, as is the case in *Crossings*, the strategic goal is further down the game-tree. *Epaminondas* blends tactical and strategic play and contains longer sequences of *mating* moves coupled with a high branching factor. This is important to note and makes *Epaminondas* more like *Chess* in complexity and play style while incorporating a long term strategy similar to *Go*.

Mean: 96.38
Standard Deviation: 27.85

Figure 5.6: Epaminondas Branching Factor Over Time.

59

5.4 Domain Comparisons

Abbott stated that his goal with *Epaminondas* was to increase the complexity of *Crossings* [2]. It has been shown that Abbott achieved his goal by expanding the boar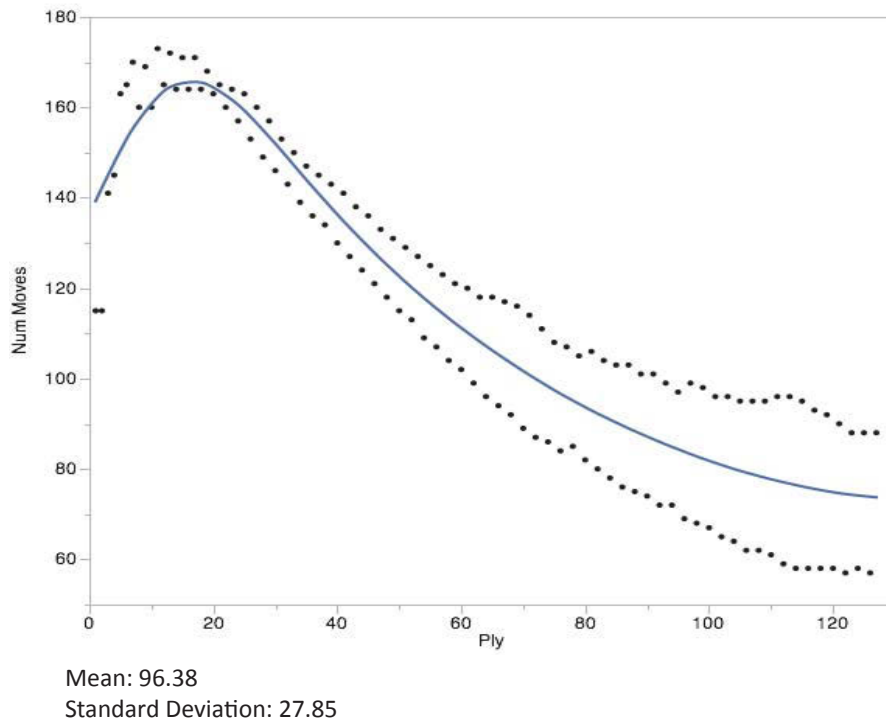d as well as allowing *crossed* pieces to move and to be captured. Usually researchers merely contract, or expand, the board size to decrease or increase the complexity of a game. Abbott shows that it is also reasonable to tweak legal moves to achieve a more complex game. This increase in game space also leads to different behaviors. While *Crossings* experiences a quick peak, then decline, of average moves available per player, *Epaminondas* has a longer build up, and then slower decline of average moves available. This behavior is due to the greater distance between players in *Epaminondas*. In *Crossings*, larger phalanxes quickly occupy enemy territory and come under enemy attack. This leads to a higher attrition rate earlier in the game. This particular trend may lead to opening gambits that could prove the game a theoretical win for White since it has the initiative from the beginning of the game and could possibly build phalanxes that push through Black's defenses. Such gambits were not observed in *Epaminondas* play. In addition, the early attrition rate may enable MC agents to play *Crossings* better since the agent possesses fewer moves to play out earlier in the game while in *Epaminondas* the number of moves available grows considerably in the first 20 ply.

5.5 Monte-Carlo Based Search

Each MC agent played against a Min-Max $\alpha\beta$ agent set to a search depth of 3. MC agents played 10,000 games (5,000 as White, 5,000 as Black) at 1, 5, 10, and 15-second time intervals in both domains.

5.5.1 Crossings.

Crossings was the first test environment and set the baseline for domain comparison. Figure 5.8 shows the win ratios for UCT and HUCT. They are compared to a base Min-Max $\alpha\beta$ agent playing as White against a Min-Max $\alpha\beta$ agent playing as Black set to a search

depth of 3. Although tested in discrete intervals, a continuous graph better illustrates the agents performance over time. Table 5.1 lists actual win rates per time interval.

Figure 5.7: Crossings White Win % (Error bars are 95% confidence interval of the mean).

Table 5.1: Crossings Win/Loss/Draw Percentages: Agents Playing as White.

	1 sec			5 sec			10 sec			15 sec		
	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw
$\alpha\beta$	0.624	0.273	0.103	0.609	0.300	0.091	0.638	0.270	0.092	0.630	0.292	0.078
UCT	0.092	0.908	0.000	0.394	0.605	0.103	0.472	0.528	0.000	0.494	0.506	0.000
HUCT	0.080	0.920	0.000	0.391	0.609	0.000	0.501	0.498	0.001	0.530	0.469	0.001

61

Table 5.2: Crossings T-Tests: Agents as White.

	Interval	T-Value	P-Value	Statistically Different?
UCT vs HUCT	1	1.701	0.092	No
UCT vs HUCT	5	0.287	0.775	No
UCT vs HUCT	10	3.193	0.0019	Yes
UCT vs HUCT	15	3.611	0.0005	Yes

ANOVA tests confirmed statistical differences existed between the three algorithms. Afterwards, Student's t-tests pinpointed where the algorithms differed significantly. Table 5.2 shows that UCT and HUCT differ in performance at the 10 and 15 second time intervals with HUCT outperforming UCT. This shows that the added heuristic guidance pays dividends at higher time intervals. The performance improvement is modest, however, with the HUCT agent performing only 3 and 4 percent better respectively. Statistical differences existed between Min-Max $\alpha\beta$ and both MC agents at all time intervals.
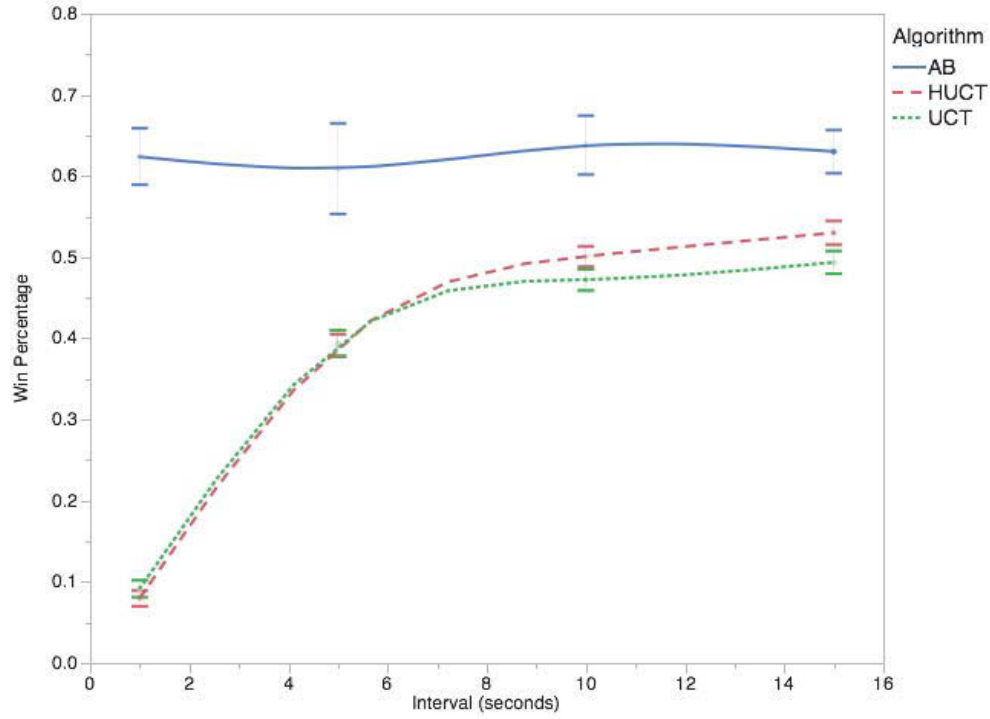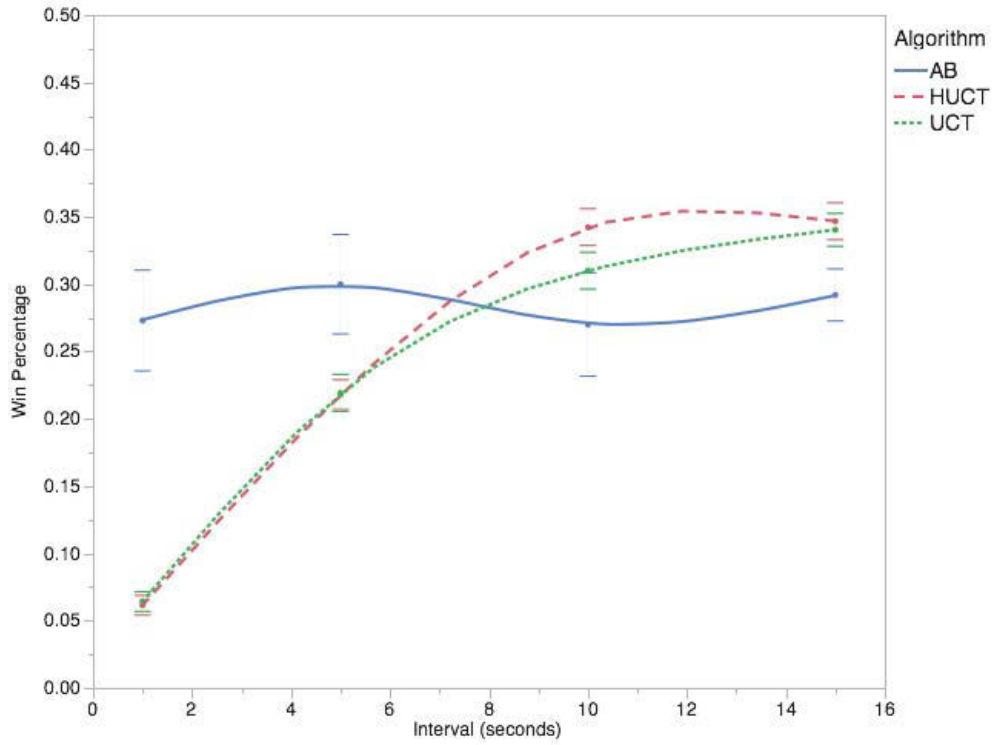
Figure 5.8: Crossings Black Win % (Error bars are 95% confidence interval of the mean).

Table 5.3: Crossings Win/Loss/Draw Percentages: Agents Playing as Black.

	1 sec			5 sec			10 sec			15 sec		
	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw
$\alpha\beta$	0.273	0.624	0.103	.0300	0.609	0.091	0.270	0.638	0.092	0.292	0.630	0.078
UCT	0.064	0.936	0.000	0.219	0.781	0.000	0.310	0.689	0.001	0.340	0.658	0.002
HUCT	0.061	0.938	0.001	0.218	0.781	0.001	0.342	0.655	0.003	0.347	0.650	0.003

Figure 5.8 shows the win ratios for UCT and HUCT. They are compared to a base Min-Max $\alpha\beta$ agent playing as Black against a Min-Max $\alpha\beta$ agent playing as White set to

Table 5.4: Crossings T-Tests: Agents as Black.

	Interval	T-Value	P-Value	Statistically Different?
UCT vs HUCT	1	0.541	0.590	No
UCT vs HUCT	5	0.137	0.891	No
UCT vs HUCT	10	3.370	0.001	Yes
UCT vs HUCT	15	0.700	0.4857	No

a search depth of 3. Again, a continuous graph better illustrates the agents performance. Table 5.3 lists actual win rates per time interval.

Student's t-tests in Table 5.4 show that UCT and HUCT only differ in performance at the 10 second interval. It is interesting that they perform very closely at the 15-second interval. One possible explanation is that the lower game-tree complexity, coupled with fewer moves as the Black player, allows UCT to complete enough simulations to overcome the gap created by the heuristic guidance HUCT receives. A secondary explanation, is that MC based methods can over correct their early estimations, resulting in poor play, followed by good play as the estimations return back to their previous values. Figures 5.9 and 5.10 display the win percentage of the MC agents based upon the number of simulations they completed. For these figures, the 5,000 trials were broken into samples of 100 games. The win rates and simulations were then averaged across the 100 game sample. The cluster of plots is indicative of the time interval. For example, the plots at 90,000 simulations is drawn from the 1 second time interval, the 200,000 simulations from the 5 second interval and so on. These figures show the average simulations completed at those time intervals and compares their win percentages across 100 game sets. This is presented to convey three points: one, UCT and HUCT produce more simulations when given more time; two, their average win ratios increase as the time interval increases, and, three, the stochastic nature of the algorithms as one can see the disparity in win ratios across 100 game samples.
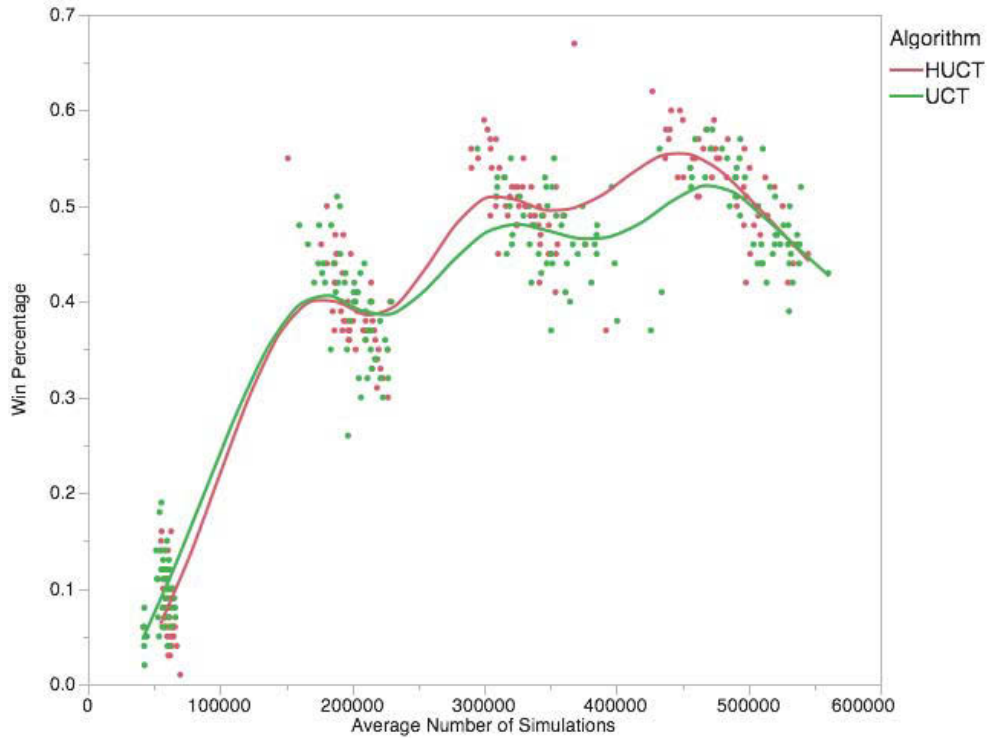
Figure 5.9: Crossings White Win % vs Average Simulations.

Both UCT and HUCT display interesting behaviors in figures 5.9 and 5.10. The increase and decrease of each algorithm's effectiveness conforms to expected MC behavior. The data indicates that MC methods have a fluid estimation of a game state. It may find a *good* move early on but, as it plays out more simulations, it may choose *bad* play outs of the state and begin underestimating it. This leads the agent to select other moves, possibly overestimating them. This results in *poor* move selection. Over time, the agent can correct its move selection back to the *better* move. A natural extension of these experiments would be to run more trials beyond the 15-second interval to see if a threshold in win percentage appears. From these graphs, both UCT and HUCT perform similarly at low simulation
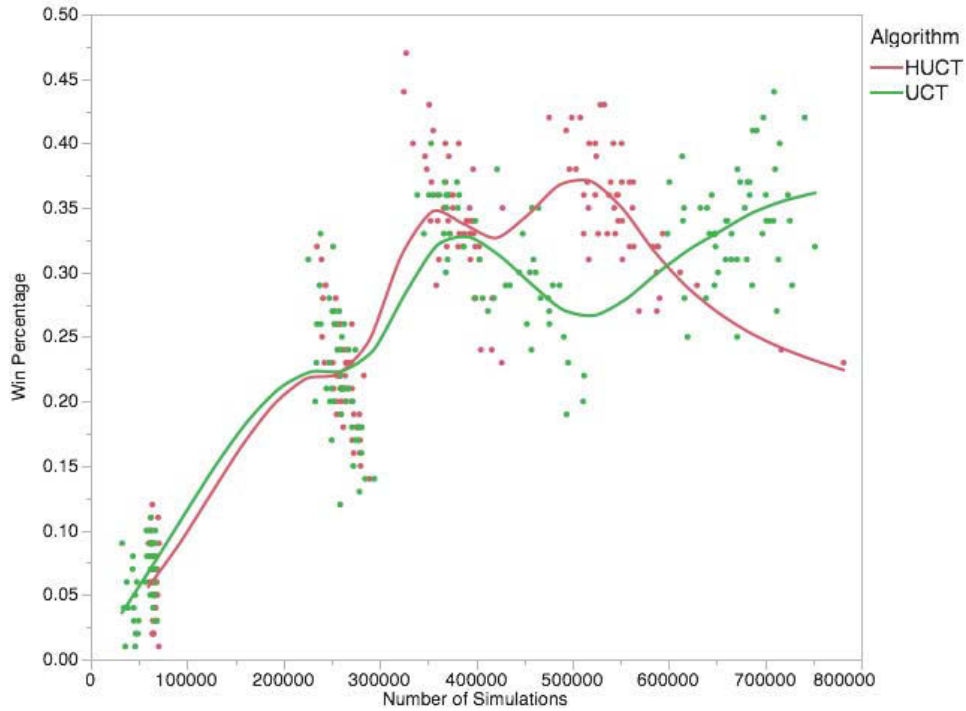
Figure 5.10: Crossings Black Win % vs Average Simulations.

intervals. As simulations broach 220,000 games as a White player, HUCT wins more often when it completes the same amount of simulations as the UCT agent. However, any noticeable gains at 220,000 disappear as the agent exceeds 500,000 simulations. It is likely that each agent hits this threshold in a losing game with very few moves left. This allows the agent to complete lots of simulations quickly, yet for nothing as it is already losing. Both algorithms appear to have similar behavior as Black, however, UCT begins to win more after 600,000 simulations. One explanation is that UCT plays a stronger game as Black and therefore wins more often near an end game state, or, this is merely the ebb and

flow phenomenon appearing once again. Comparing the agents win percentage to game length offers another explanation.

Figures 5.11 and 5.12 show the relative strength of each agent as a function of game length. Both agents appear more likely to win as the game length increases. As a White player, if the HUCT agent exceeded the 26th turn of the game, it became more likely to win than a UCT agent. The same is true for the 23rd turn as a Black player. However, game length is not always indicative of success. It is merely noted that longer games seem to favor the HUCT algorithm possibly indicating stronger end game play as the number of available moves diminishes, allowing HUCT to find *good* moves near the end of the game. However, from a simulation standpoint, the HUCT algorithm does poorly as the agent achieves 500,000 simulated games. It is a fascinating paradox then, where the agent wins more often as the game length increases, yet appears to lose in game states where it can complete a high number of simulations. It may be that HUCT wins more games through late game crossings with more pieces remaining on the board while UCT wins more games through attrition. This means the heuristic guided node selection creates an agent that plays strategically different moves than its UCT counterpart.
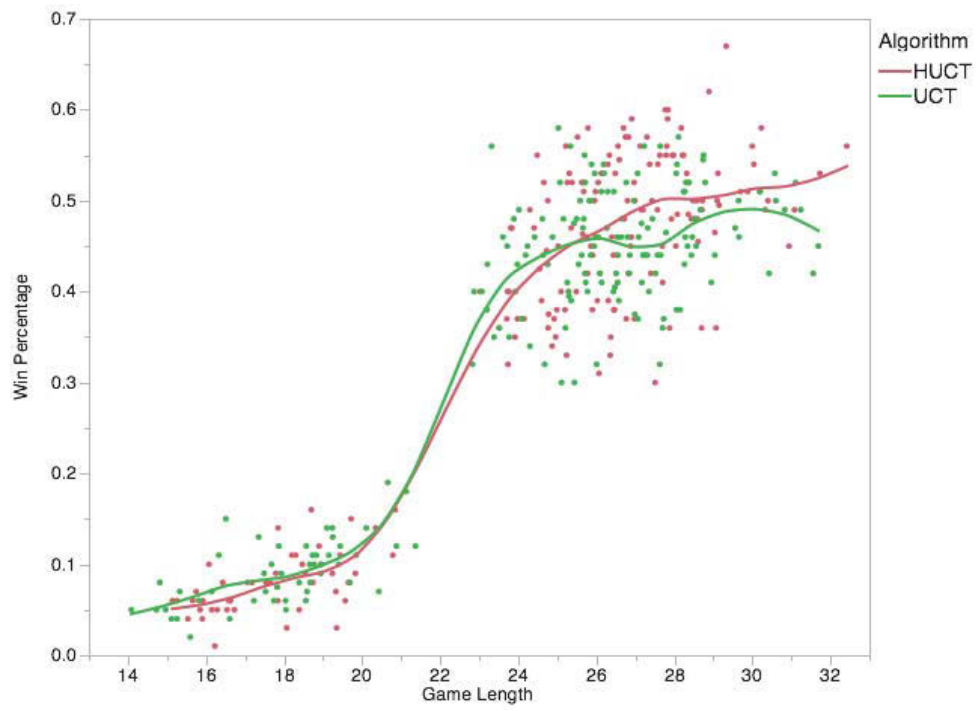
Figure 5.11: Crossings White Win % vs Game Length.
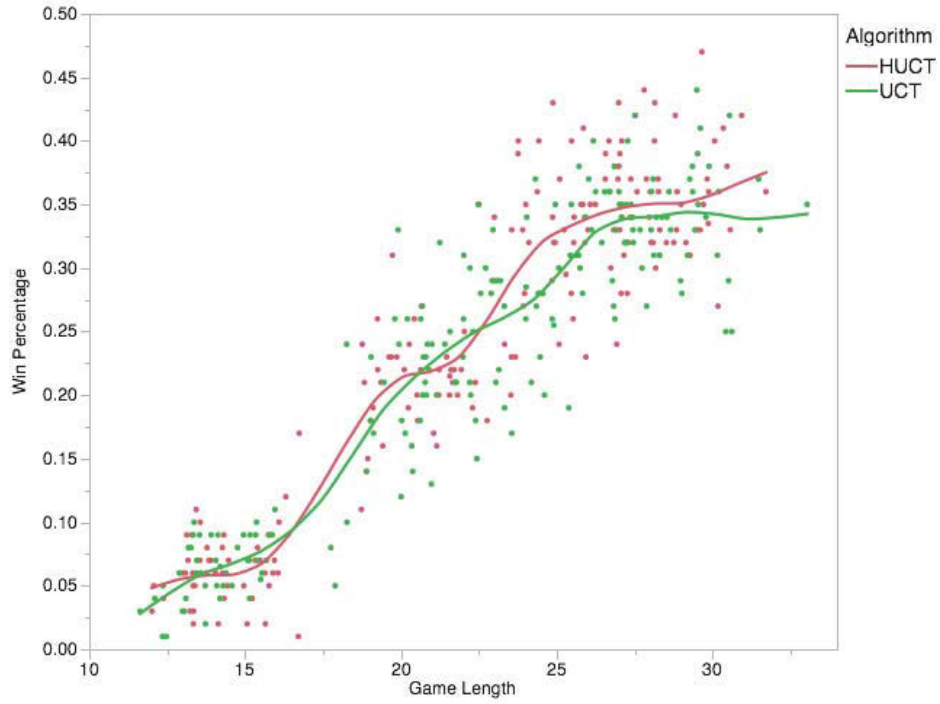
Figure 5.12: Crossings Black Win % vs Game Length.

5.5.2   *Epaminondas.*

The agents were tested in the higher complexity domain of *Epaminondas*. The MC agents and time intervals remained the same. Only the domain changed.
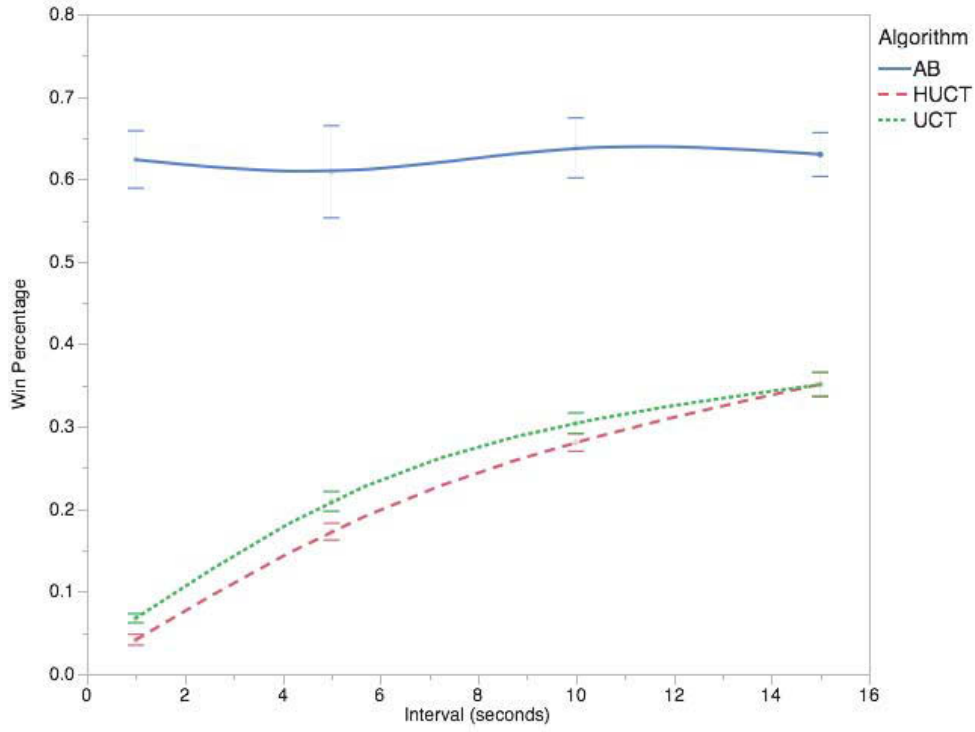
Figure 5.13: Epaminondas White Win % (95 % Confidence Interval of the Mean).

Table 5.5: Epaminondas Win/Loss/Draw Percentages: Agents Playing as White.

	1 sec			5 sec			10 sec			15 sec		
	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw
$\alpha\beta$	0.484	0.459	0.057	0.596	0.403	0.001	0.593	0.406	0.001	0.596	0.401	0.003
UCT	0.068	0.932	0.000	0.209	0.791	0.000	0.304	0.696	0.000	0.351	0.649	0.000
HUCT	0.042	0.958	0.000	0.173	0.827	0.000	0.281	0.719	0.001	0.351	0.648	0.001

Figure 5.13 and Table 5.5 show the win ratios for UCT and HUCT. ANOVA tests indicated differences existed between the three algorithms. Students t-tests (Table 5.6)

70

Table 5.6: Epaminondas T-Tests: Agents as White.

	Interval	T-Value	P-Value	Statistically Different?
UCT vs HUCT	1	6.105	0.000	Yes
UCT vs HUCT	5	4.701	0.000	Yes
UCT vs HUCT	10	2.876	0.005	Yes
UCT vs HUCT	15	0.039	0.969	No

confirmed that UCT outperformed HUCT at every time interval except 15 seconds. In addition, both algorithms fell far short of their win percentages in *Crossings*. UCT's performance declined by an average of 10 percent across all intervals. HUCT's performance fell by 13 percent across the same intervals. Clearly, both algorithms' performance declined as they went from a lower game-tree complexity to a higher one. Table 5.6 shows the Student's t-tests results for both agents playing as White. The results indicate that in a large game-tree space, evaluating the board state significantly impacts HUCT's performance. The loss of simulated games adversely affects the agent. Only in the 15-second time interval is it able to catch up with the UCT agent. These findings indicate that the heuristic function for *Epaminondas* needs further refinement. A better evaluation of the board state will lead HUCT to more promising parts of the search space. In addition, the decision complexity may also impact its performance. With a large number of moves to evaluate, the agent may not have enough time to validate a chosen move through simulated games. A combination of heuristic refinement as well as speeding board state evaluations may overcome these issues. Additionally, longer time intervals, 30 to 60 seconds per move, could prove fruitful.

Figure 5.14 and Table 5.7 show that when playing Black, HUCT performed more closely to UCT. Student's t-tests confirm that the UCT algorithm only outperforms HUCT at the lower time intervals of 1 and 5 seconds. HUCT performs as well as UCT in the 10

and 15 second intervals in this scenario. One possible reason for this behavior comes from the aggressive nature of the Min-Max $\alpha\beta$ *Epaminondas* agent. If UCT and HUCT thwart early attacks, the game space grows smaller through piece attrition. This allows both agents to perform more simulations and find *better* moves. For HUCT, this means the heuristic function allows it to pull even with UCT although it is still performing fewer simulations per turn. However, they both fail to overcome Min-Max $\alpha\beta$'s advantage.
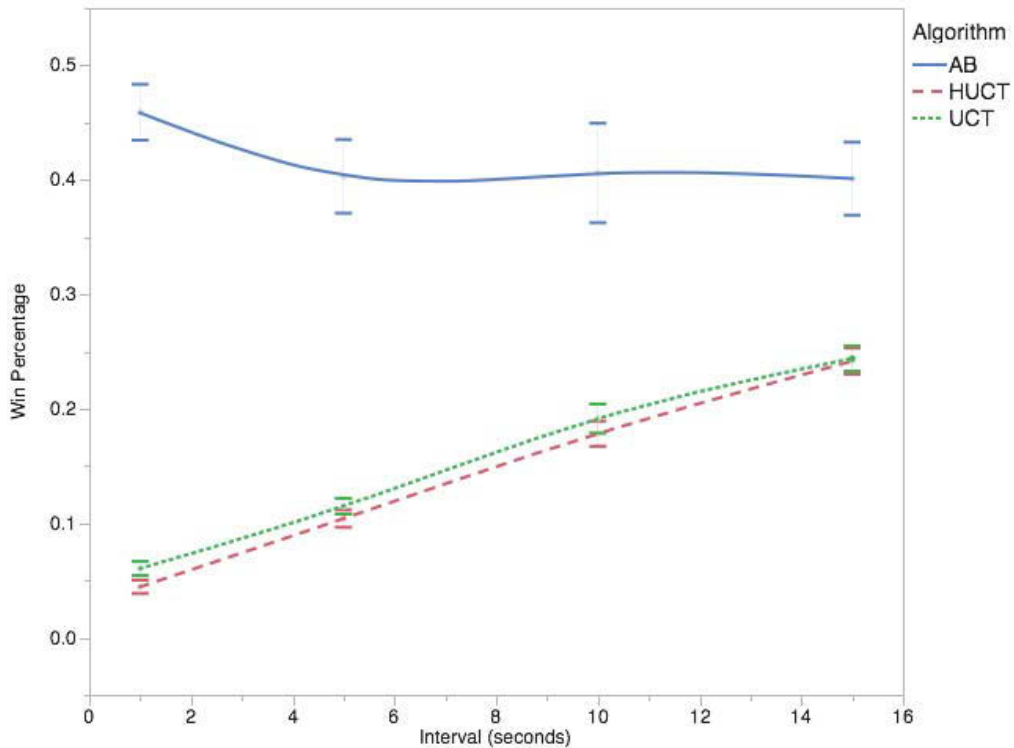

Figure 5.14: Epaminondas Black Win % (95% Confidence Interval of the Mean).

Figures 5.15 and 5.16 show how the agents' simulations compared with their win percentage. As with *Crossings*, the 5,000 games were broken into 100 game samples

Table 5.7: Epaminondas Win/Loss/Draw Percentages: Agents Playing as Black.

	1 sec			5 sec			10 sec			15 sec		
	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw	Win	Loss	Draw
$\alpha\beta$	0.459	0.484	0.057	0.403	0.596	0.001	0.406	0.593	0.001	0.401	0.596	0.003
UCT	0.060	0.940	0.000	0.115	0.885	0.000	0.191	0.809	0.000	0.244	0.756	0.000
HUCT	0.044	0.956	0.000	0.104	0.896	0.000	0.178	0.822	0.000	0.242	0.758	0.000

Table 5.8: Epaminondas T-Tests: Agents as Black.

	Interval	T-Value	P-Value	Statistically Different?
UCT vs HUCT	1	3.809	0.0002	Yes
UCT vs HUCT	5	2.112	0.037	Yes
UCT vs HUCT	10	1.579	0.118	No
UCT vs HUCT	15	0.2773	0.7822	No

across the 1, 5, 10, and 15 second time intervals to calculate average simulations and win percentages. In the *Epaminondas* domain, HUCT plays evenly with UCT beginning at 50,000 simulations per turn when playing White. A similar trend is seen in Figure 5.16 with both experiencing drop offs at 120,000 and 140,000 simulations respectively. Although HUCT did not win more often than UCT, it clearly plays about as well even though the domain is much larger than *Crossings*. The crossover point for success is at 15 seconds, and one could extrapolate that HUCT may perform better than UCT at a 30 second interval in this domain as the White player. We cannot make the same conclusion about playing Black since both algorithms experience a steep decline in win percentage after 140,000 simulations per turn. Additionally, the sharp reduction in simulations is readily apparent between the two domains with neither agent coming close to the 200,000 and 500,000 simulated games accomplished playing *Crossings*.
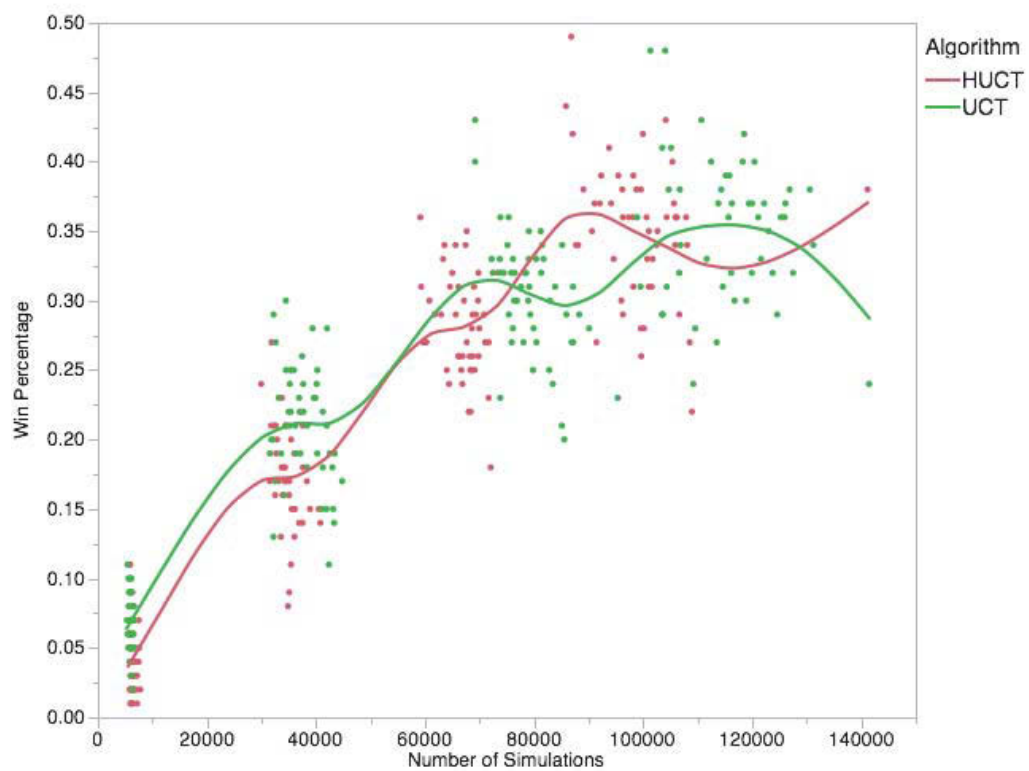
Figure 5.15: Epaminondas White Win % vs Sims per Ply.
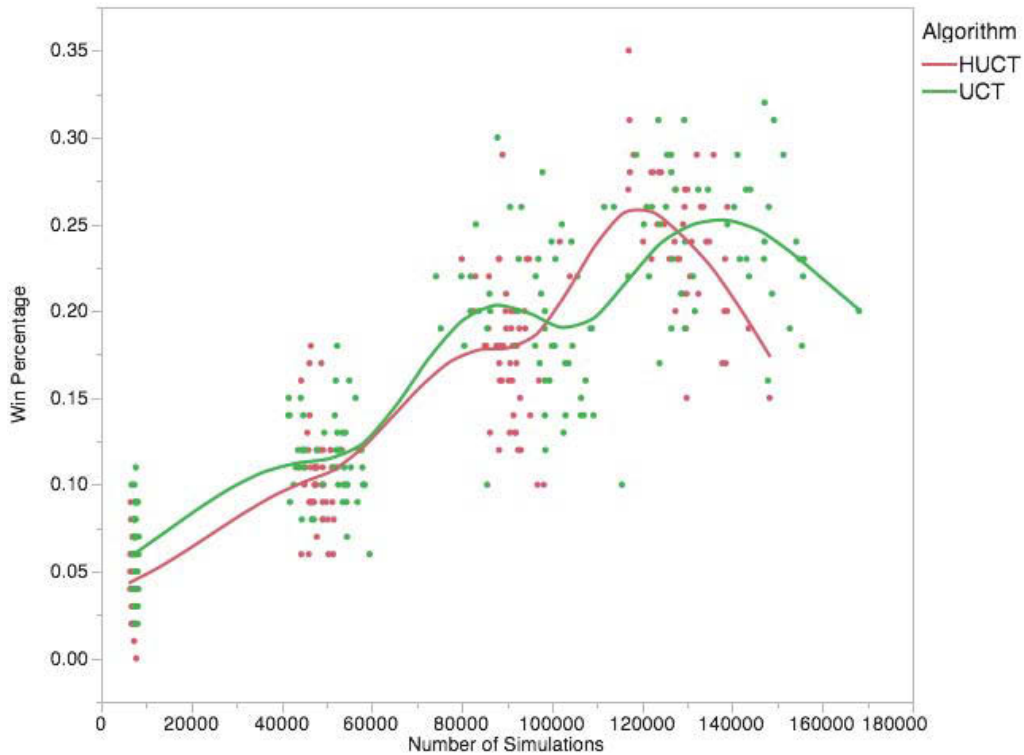
Figure 5.16: Epaminondas Black Win % vs Sims per Ply.

Figures 5.17 and 5.18 relay very interesting details. It appears that as the game goes longer, both agents appear more likely to lose. This is especially true for White after turn 70 and Black around turn 56. The average length of an *Epaminondas* game is 56 with a standard deviation of 4. This indicates that the longer games are indicative of either Min-Max $\alpha\beta$ having to relaunch attacks after many failed ones, still ultimately winning, or that neither MC based algorithm performs well in a higher tactical environment where there are disparate pieces left on the game board. In this situation, the agent may rate a singleton phalanx advancing towards the enemy back row higher than collecting pieces together into a larger phalanx, two to three moves down the decision tree. In these situations Min-

Max $\alpha\beta$ has the upper hand since it evaluates its next few moves, thus building larger, more effective phalanxes able to cross the board more rapidly than the singletons moved by either MC agent. The average length of games between all three agents indicates that the earlier evaluation of the average length of an *Epaminondas* game may be too low. Again, further refinement of the *Epaminondas* heuristic function could solidify this observation.
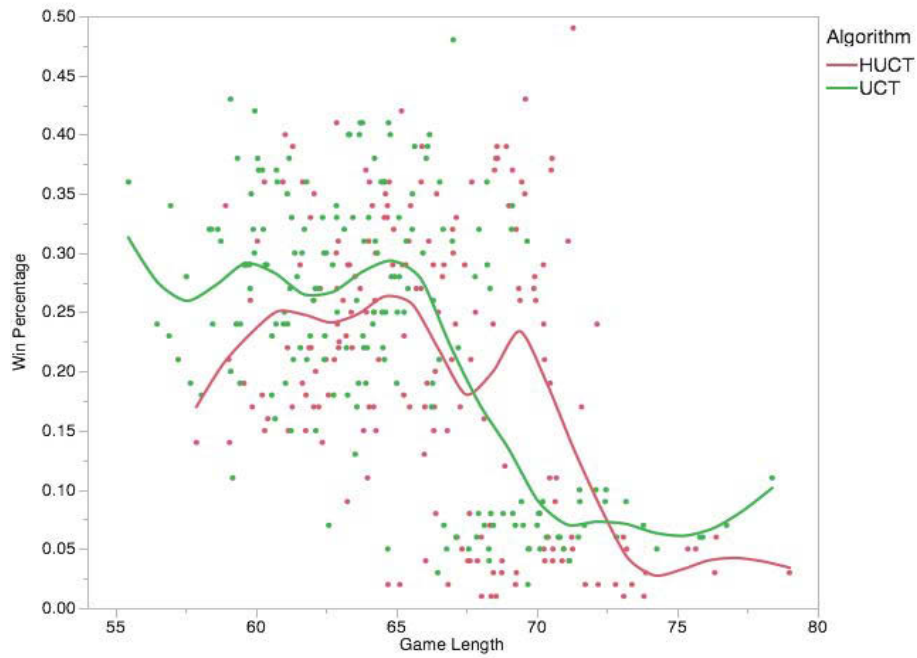
Figure 5.17: Epaminondas White Win % vs Game Length.
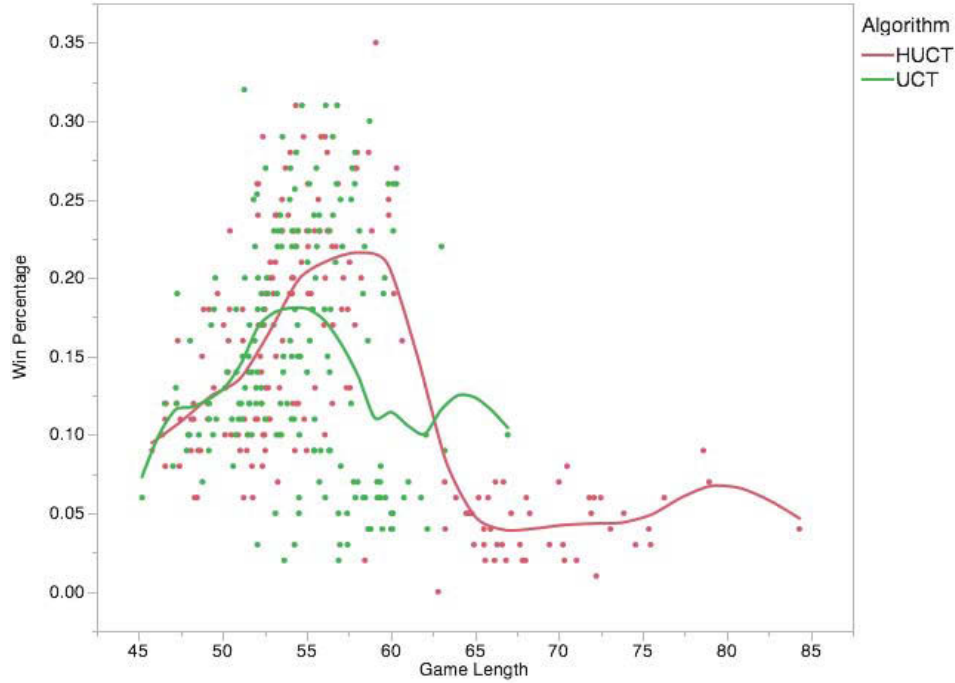
Figure 5.18: Epaminondas Black Win % vs Game Length.

5.6 Observations

According to the data, MC based agents fair better in the smaller search domain of *Crossings*. However, there is more to their failure in *Epaminondas* than sheer game-tree complexity. If game-tree complexity alone affected MC agents, then they would fail spectacularly in *Go*, which is not true. This indicates the possibility that complex moves and mating combinations may temper MC effectiveness. If a game has a high state-space and game-tree complexity, yet consists of smooth board transitions, then MC based agents can perform well [15]. A game of *Go* does not end abruptly and board transitions are seldom drastic. The same cannot be said of *Chess* in which games can end abruptly and

77

where favorable board positions can instantly become untenable. According to [36] this is due to *trap* states existing in *Chess* where an Min-Max $\alpha\beta$ agent is able to identify a shallow trap (a mate in three for example) while MC agents fail to estimate such a state correctly. Others have noted similar issues in tactical board positions in *Hex* [9] and even in some *Go* board states [60]. Winands provides a similar analysis of UCT failures in the domain of Lines of Action (LOA) [57]. The drop off in effectiveness of both MC based agents in *Epaminondas* indicates the game contains tactical states which a Min-Max $\alpha\beta$ agent can correctly navigate while MC based agents do not.

The puzzles presented in Chapter 2 for *Epaminondas* serve as prime examples of tactical board states that can arise in the game. Both MC based agents were given these board positions with a 6 hour time limit. Neither produced the winning move for any of the puzzles while the Min-Max $\alpha\beta$ agent correctly solved each one with puzzle three taking 3 hours to solve while it was able to solve the others in under a minute. These puzzles serve as an indicator of how smaller tactical states can heavily affect the outcome of a game, yet, even in these smaller sub-domains, MC agents failed to achieve success. However, the agents did perform well in *Crossings*, realistically just a smaller version of *Epaminondas*, which contains similar tactical positions, begging the question: why?

One, *Crossings* is a smaller domain and has a smaller decision tree. At the first ply, it has one hundred less states to choose from than *Epaminondas* and this game-tree space grows exponentially smaller than *Epaminondas* at each ply down the tree. The data shows that the MC agents do better as they complete more game simulations. In *Epaminondas* they completed far fewer simulations at each time interval resulting in poorer play. This indicates that game-complexity plays a role in the effectiveness of MC agents. However, as one increases the time interval, MC agents can slowly overcome the game-complexity issue as they can complete more simulations and derive better estimates for *good* moves. Expanding the time interval to 30, 45, and 60 second trials could show

78

MC agent improvement in *Epaminondas*. A second reason for poor play is the distance between the players in *Epaminondas*. In *Crossings*, the games are shorter with an increased likelihood of attack and piece attrition. Less pieces on the board results in more simulations completed by the MC agent. Barring a super-aggressive start, attacks come later in a game of *Epaminondas*, approximately 20 moves into the game, allowing less simulations for the MC agents while playing into the tactical strength of Min-Max $\alpha\beta$ agents. The coupling of a high decision complexity with a high frequency of tactical subgames could be a contributor of MC failure. Failure that can be overcome through the introduction of heuristic knowledge. Lorentz [32], Winands [56], and Lorentz and Horey [33] have all shown that the introduction of heuristic knowledge into a MC agent can positively affect its ability to counter Min-Max $\alpha\beta$ tactical strength in the games of *Amazons*, *Lines of Action* and *Breakthrough*. HUCT's performance in *Crossings* shows similar results. As noted earlier, with more time, and a better heuristic function, similar results may be achieved in *Epaminondas*. For now, *Epaminondas* remains in the domain of Min-Max $\alpha\beta$.

5.7 Summary

This chapter analyzed the data from each of the three experiments outlined in Chapter 4. First, it summarized how agents for each domain were constructed. Then, for the first time in Artificial Intelligence (AI) literature, it presented the state-space and game-tree complexities for *Crossings* and *Epaminondas*. Additionally, it proved that both games are unsolvable by current methods. This chapter also analyzed the effectiveness of two MC agents, UCT and HUCT, across both domains. It showed that adding heuristic knowledge to a MC agent positively affects its play in the smaller domain of *Crossings* but negatively affects its play in the higher complexity domain of *Epaminondas*. The chapter also analyzed how game length and simulations accomplished by the MC agents compared with win ratios to gather confidence in these findings. These results showed that the heuristic function for *Epaminondas* needs further refinement as both the number of simulations

and game length appeared to have a negative impact on both MC algorithms. Finally, by analyzing the data presented, this chapter proposed reasons why MC agents may fail in games such as *Hex* and *Chess* and identified *Epaminondas* as another domain where Min-Max $\alpha\beta$ agents currently dominate MC methods.

VI. Conclusions

This chapter begins by answering each research question in detail and provides observations derived from the experimental data (Sections 6.1-6.5). Afterwards, Section 6.6 provides general conclusions about each game domain. Finally, Section 6.7 finishes the chapter outlining areas of future work.

6.1 Are *Crossings* and *Epaminondas* solvable?

Through building two game playing agents using the Min-Max with Alpha-Beta ($\alpha\beta$) pruning algorithm, the state-space and game-tree complexity for *Crossings* are 10^{23} and 10^{81} respectively. *Crossings* outstrips *Lines of Action* and *Checkers*; games played on the same size board. *Epaminondas'* state-space and game-tree complexities are 10^{61} and 10^{137}, placing it above well researched games such as *Othello*, *Hex 11x11* and *Chess*. However, it falls short of *Go*.

At this time, neither game appears solvable by current methods. However, during the analysis of *Crossings* the White player seemed to have a slight advantage in all the games played. Results indicate strong opening sequences exist that may prove unbeatable. If they do exist, then *Crossings* may prove solvable although it has high state-space and game-tree complexities. *Epaminondas* did not display such behavior since neither player can launch large, concerted attacks into each other's territory until at least five to ten turns have elapsed. The distance between the players appears to lead to more balanced play with neither player having a distinct advantage over the other.

6.2 Does move complexity impact game complexity?

The unique moves available to a player in both *Crossings* and *Epaminondas* affect the complexity of the game. The analysis of *Crossings* distinctly shows how its unique movement rules push its complexity beyond well known games such as *Lines of Action*.

Its complexity hinges on the number of moves available to a player each turn. Analysis of move complexity also indicates that both games are tactical in nature with *Epaminondas* incorporating both long term strategy as well as tactical move sequences which may have led to the poor performance of Monte-Carlo (MC) agents in this domain. Although very similar to *Epaminondas*, MC agents faired better in *Crossings*, even surpassing the Min-Max $\alpha\beta$ agent playing Black. However, the MC algorithms lagged severely behind Min-Max $\alpha\beta$ agents in *Epaminondas* at all time intervals.

6.3 With respect to MC-based search algorithms such as Upper Confidence Bounds Applied to Trees (UCT), does game complexity impact the algorithm's performance?

Both algorithms performed poorly as they moved from *Crossings* to *Epaminondas*. This behavior is akin to Min-Max $\alpha\beta$ performance shortfalls as the environment grows in complexity where heuristic evaluation becomes vitally important in pruning the search space. However, MC success in *Go* prevents making the assumption that only game-tree complexity plays a role in MC success or failure. As discussed previously, MC success may hinge more on game state transition smoothness, an idea discussed in [19] concerning the UCT algorithm. *Go* contains simple move generation: placing one stone at a time with resulting captures of connected stones. Stones never move positions and, although captures occur, the transition between states is smooth (little changing between them). This does not hold true in *Crossings*, *Chess* and *Epaminondas*, and may indicate that producing expert level agents for *Go* on a 19 x 19 board may not be the proverbial "holy grail." Instead, researchers may find that hybrid techniques are necessary to solve tactical games whose game-tree complexities are far less than *Go*'s.

6.4 Does adding heuristic knowledge to UCT improve its performance?

Adding heuristic knowledge to a MC based search agent can positively impact the agents performance but only for the smaller of the two games presented. For *Epaminondas*, regular UCT outperformed Heuristic Guided UCT (HUCT) at almost every interval. The branching factor for *Epaminondas* affected the performance of the algorithm as well as a lack of game knowledge. Even the Min-Max $\alpha\beta$ agent shows worse performance in the game and any setting beyond a search depth of 3 resulted in Min-Max $\alpha\beta$ exceeding the time threshold and returning *bad* moves. The heuristic function for *Epaminondas* needs further improvement in order to prove the value of the HUCT algorithm in a large search space. Additionally, further refinement of the heuristic evaluation function can improve the game-tree complexity estimation.

6.5 Do UCT and HUCT perform better as time intervals increase?

Although each algorithms' performance dropped when moving from *Crossings* to *Epaminondas* each agent's win percentage increased as the time interval expanded. These results fall in line with current research and the expected behavior of the algorithms. Both UCT and HUCT perform better as they accomplish more simulations per turn. When given more time, each can perform more simulations, which enables them to make better decisions about the game states. Future work can determine if a simulation threshold exists for both algorithms with respect to each domain. If a threshold is reached, and the win ratio does not meet or exceed Min-Max $\alpha\beta$'s win ratio, then one could conclude that the domain thwarts MC based agents.

6.6 General Conclusions

Crossings and *Epaminondas* provide interesting testing environments for future work. The steep tactical slant of each game, and their complex movements, open a promising area of research where decision complexity is crucial to the problem environment. One

can easily correlate high branching factors with complex real-world environments where an agent may face multiple decisions at once. Generic MC methods may prove inadequate in solving such problems. However, heuristic based MC methods may fair better. Although MC methods have shown promise in General Game Playing (GGP), there may be a limit to their effectiveness. The games selected for GGP range in complexity but do have general heuristic knowledge (such as the concept of mobility) added to them [20]. Of course, the focus is on developing agents that can play any game and MC methods may prove vital to reaching this goal. The ability to build game playing agents with zero game knowledge is impressive and worthy of future study. In addition, as hardware and computational power increases, so will the effectiveness of MC based search algorithms.

6.7   Future Work

The interesting phalanx maneuvers encountered in the games of *Crossings* and *Epaminondas* generate new, and unique, game domains. This twist to the games makes them stand out from other two-player strategy games where pieces may value high mobility and centrality akin to *Lines of Action* or a combination of such ideas in *Chess*. However, these games never allow multiple pieces to move together and capture multiple opposing pieces at once. Although this may seem very subtle, it has a huge impact on the game and the algorithms playing it. For one, move generation becomes more time consuming. This costs algorithms such as Min-Max, Min-Max Alpha-Beta, and Monte-Carlo based agents. The longer move generation takes, the less time an agent has to search. One important breakthrough for both games can come through ingenious ways to speed move generation. Furthermore, the analysis of the board for captures and movements may be compressed. With these modifications, agents for both domains will become stronger in the future since they can search deeper and faster in the same amount of time.

Another avenue of research resides in tuning the heuristic evaluation function for these games, especially *Epaminondas*. One may find *Epaminondas'* complexity to be greater

than currently estimated. Heuristic refinement will also assist the HUCT algorithm. Results showed that the HUCT agent was able to play on the same level, if not better, than the basic UCT algorithm with minimal code modification. Heuristic development for *Crossings* and *Epaminondas* is just beginning. After analyzing thousands of self-play games, there are multiple areas of interest. For example, a weighted graph of the most played portions of an *Epaminondas* board showed an extremely high crossing rate at columns D and K. One could encode defense of these columns into the heuristic function. In addition, it appears White has an advantage in most of the games, consistently winning more often than Black. Additionally, there are certain patterns that arise during play for defensive measures. Incorporating these into a pattern database could increase the speed of heuristic evaluation. The heuristic function can be further tuned with real-time learning algorithms such as TD-Lambda or possibly even through Gaussian estimation. Both of these methods require the agent to learn weighted factors for the heuristic functions used by the agents to evaluate the board state. These weights can reduce the search space for Min-Max $\alpha\beta$ agents, resulting in faster response times and better move selection. This could lead to better game complexity estimates as well as stronger HUCT agents.

Finally, one can test the HUCT algorithm in other game environments. Games such as *Lines of Action* and *Hex* may be well suited for testing since they are heavily researched and strong heuristic evaluation functions exist for each game. Additionally, the complexity of *Lines of Action* is slightly smaller than *Crossings* and could be used to verify the results presented. Finally, incorporating the RAVE concept of slowly dwindling the weight of the heuristic value in HUCT is another promising avenue of work. This means, that as the algorithm completes more simulations, the heuristic value becomes less important, relying more on the estimated play out results.

Appendix: Appendix A

Table A.1: Number of Possible Positions Per Pieces on Board for Crossings

# Pieces	# Positions
2	4,032
3	329,280
4	10,161,984
5	243,980,352
6	4,798,355,520
7	79,515,668,544
8	1,133,098,330,176
9	14,100,779,266,112
10	155,108,571,967,552
11	1,522,884,161,171,520
12	13,452,143,423,713,344
13	107,617,147,389,734,976
14	784,067,788,125,237,312
15	5,227,118,587,501,604,928
16	32,016,101,348,447,350,848
17	180,794,048,874,410,966,400
18	944,024,242,258,584,876,480
19	4,568,396,469,282,496,744,320
20	20,519,761,083,204,457,478,880
21	85,589,233,763,681,006,029,440
22	331,306,115,164,919,217,610,560
23	1,188,092,899,283,252,528,764,800
24	3,936,420,640,385,192,265,984,720
25	12,006,641,574,836,196,656,135,040
26	33,561,989,414,523,512,489,171,520
27	85,476,562,476,935,603,489,953,920
28	196,729,524,470,922,506,518,623,840
29	403,931,343,020,145,070,418,467,200
30	722,588,291,402,703,959,304,146,880
31	1,068,173,995,986,605,852,884,391,040
32	1,101,554,433,361,187,285,787,028,260
Total:	3,629,590,441,722,350,978,583,241,845

Table A.2: Number of Possible Positions Per Pieces on Board for Epaminondas

# Pieces	# Positions
2	28,056
3	6,181,672
4	512,274,504
5	33,607,019,880
6	1,825,982,909,800
7	84,516,924,524,136
8	3,401,806,213,197,672
9	120,953,109,803,553,384
10	3,846,308,891,753,861,736
11	110,493,964,526,748,078,696
12	2,891,258,738,449,908,765,288
13	69,390,209,722,797,811,012,200
14	1,536,497,501,004,808,673,006,184
15	31,549,415,353,965,404,752,941,672
16	603,382,568,644,588,365,900,517,992
17	10,789,900,051,056,168,425,515,618,920
18	181,030,545,301,053,492,472,540,271,208
19	2,858,377,031,069,265,670,619,057,328,744
20	42,589,817,762,932,058,492,223,954,588,264
21	600,313,621,801,328,062,557,061,455,516,264
22	8,022,372,945,890,475,017,808,003,087,700,584
23	101,849,256,530,435,595,878,258,126,157,222,504
24	1,230,678,516,409,430,116,862,285,691,066,749,544
25	14,177,416,509,036,634,946,253,531,161,089,249,896
26	155,951,581,599,402,984,408,788,842,771,982,029,416
27	1,640,379,599,045,572,132,299,853,012,860,848,280,168
28	16,520,965,961,816,119,332,448,519,629,527,115,076,200
29	159,512,774,209,510,449,373,417,899,607,583,356,545,600
30	1,478,151,627,829,905,397,705,378,180,929,378,783,244,896
31	13,160,312,387,165,439,455,997,260,070,569,634,846,284,736
32	112,684,938,948,304,476,087,649,495,004,331,338,372,513,742
33	928,789,296,086,692,530,519,176,920,862,542,483,594,851,168
34	7,375,475,915,302,998,692,501,956,288,517,719,836,643,567,200
35	56,470,651,473,862,610,411,420,178,883,887,800,079,091,585,536
36	417,173,698,928,978,359,744,784,031,104,647,568,484,589,498,656
37	2,975,279,577,358,930,690,954,095,501,794,704,303,044,454,619,712
38	20,495,187,576,348,853,903,247,506,312,010,856,710,421,790,390,368
39	136,399,210,053,397,895,119,460,781,542,139,053,935,605,577,425,600
40	877,083,619,544,198,810,863,898,927,841,472,912,163,030,543,338,872
41	5,448,417,679,883,464,573,520,299,120,385,662,559,492,088,662,863,872
42	32,682,863,369,061,187,394,280,482,185,676,987,909,292,552,027,976,704
43	189,188,862,937,091,814,961,146,511,343,601,808,647,799,839,402,756,096
44	1,055,815,421,478,083,952,425,192,010,850,820,279,952,209,311,619,520,000
45	5,673,819,140,221,690,144,249,260,636,953,436,458,594,202,126,690,163,200
46	29,317,016,584,100,626,914,184,154,105,221,381,391,940,753,482,616,979,200
47	145,395,713,795,519,275,461,335,067,886,983,645,921,567,498,361,554,470,400
48	690,613,288,111,764,077,966,511,192,341,086,331,236,472,494,328,732,493,600
49	3,133,249,789,630,968,269,315,497,775,187,618,959,484,519,783,359,054,400,000
50	13,529,568,060,420,736,496,851,395,956,424,095,325,851,628,853,301,523,886,080
51	55,326,008,764,875,843,447,075,975,428,063,257,126,722,828,560,574,705,146,880
52	212,639,164,036,988,771,847,662,625,404,939,619,560,001,844,010,535,661,512,960
53	758,526,038,455,408,694,108,063,032,765,684,383,214,038,468,611,614,683,210,240
54	2,452,983,725,183,300,406,529,161,316,522,772,199,488,471,453,706,284,795,155,200
55	6,820,491,333,436,493,813,276,204,636,185,269,042,480,140,139,573,572,357,260,800
56	13,762,777,154,970,067,873,218,055,783,730,989,317,861,711,353,068,101,363,758,400
Total:	24,080,278,526,707,819,549,851,463,172,086,945,496,984,186,580,573,839,562,868,527

Bibliography

[1] Abbott, Robert. "Under the Strategy Tree", 1975. URL http://www.logicmazes.com/games/tree.html.

[2] Abbott, Robert. "Epaminondas", December 2010. URL http://www.logicmazes.com/games/epam.html.

[3] Allis, Louis Victor. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, University of Maastricht, September 1994.

[4] Allis, Louis Victor, H. Jaap van den Herik, and M. Huntjens. *Go Moku and Threat-Space Search*. Technical report, University of Maastricht, 1993.

[5] Arneson, Broderick, Ryan Hayward, and Philip Henderson. "Mone Carlo Tree Search in Hex". *IEE Transactions on Computational Intelligence and AI in Games*, 2(4):251–258, 2010.

[6] Ashraf Abdelbar, Sara Mitri, Sherif Ragab. "Applying Co-evolutionary Particle Swarm Optimization to the Egyptian Board Game Seega". *Proceedings of the First Asian-Pacific Workshop on Genetic Programming*, 9–15. Canberra, Australia, 2003.

[7] Bouzy, B. and B. Helmstetter. "Monte Carlo Go Developments". Heinz, Herik, and Iida (editors), *10th Advances in Computer Games*, 159–174. Kluwer Academic Publishers, 2003.

[8] Boyle, Alan. "10 Most Important Board Games in History". URL http://listverse.com/2013/01/20/10-most-important-board-games-in-history/.

[9] Browne, Cameron. "Problem Case for UCT". *IEEE Transactions on Computational Intelligence and AI Games*, 5(1):69–74, March 2013.

[10] Browne, Cameron, Daniel Whitehouse, Peter Cowling, and Spyridon Samothrakis. "A Survey of Monte Carlo Tree Search Methods". *IEEE Transactions on Computational Intelligence and AI Games*, 4(1):1–43, March 2012.

[11] Brügmann, Bernd. "Monte Carlo Go". Computer Go Magazine, October 1993.

[12] Cazenave, T. "Abstract Proof Search". *Lecture Notes in Computer Science*, 2063:40–55, 2001.

[13] Cazenave, Tristan. "Multiplayer Go". *Computer Games 2008*, 50–59. LNCS 5131, 2008.

[14] Chaslot, Guillaume M.J-B., Mark H.M. Winands, and H. Jaap van den Herik. "Parallel Monte-Carlo Tree Search". *Computer Games*, 60–71. 2008.

[15] Coquelin, P.A. and R. Munos. "Bandit Algorithms for Tree Search". *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence*, 67–74. Vancouver, Canada, 2007.

[16] Cormen, Thomas, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2003.

[17] Coulom, Rémi. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search". *In: Proceedings Computers and Games 2006*, CG'06, 72–83. Springer-Verlag, Berlin, Heidelberg, 2006.

[18] Enzenberger, Markus. *Fuego – An Open-source Framework for Board Games and Go Engine Based On Monte-Carlo Tree Search*. Technical report, University of Alberta, Edmonton, Alberta, May 2009.

[19] Gelly, Sylvain, Levente Kocsis, David Silver, and Csaba Szepesvári. "The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions". *Communications of the ACM*, 55(3):106–113.

[20] Genesereth, Michael and Yngvi Bjornsson. "The International General Game Playing Competition". *Artificial Intelligence*, 107–111, 2013.

[21] Griffith, Arnold K. "Empirical Exploration of the Performance of the Alpha Beta Tree Searching Heuristic". *IEEE Transactions on Computers*, c-25(1):6–10, January 1976.

[22] Handscomb, Kerry. "Epaminondas . . . a game of classical elegance", 2000. URL www.logicmazes.com/games/epam/index.html.

[23] Henderson, Philip, Broderick Arneson, and Ryan Hayward. "Solving 8 x 8 Hex". *International Joint Conference on Artificial Intelligence*, 505–510. IJCAI, 2009.

[24] van den Herik, H. Jaap, Jos W.H.M. Uiterwijk, and Jack van Rijswijck. "Games solved: Now and in the future". *Artificial Intelligence*, 134:277–311, 2002.

[25] Hollander, Yaron and Joseph Prashker. "The applicability of non-cooperative game theory in transport analysis". *Transportation*, 33:481–496, 2006.

[26] IBM. "Deep Blue", March 2012. URL http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/.

[27] Kishimoto, Akihiro. *Transposition Table Driven Scheduling for Two-Player Games*. Master's thesis, University of Alberta, 2002.

[28] Kloetzer, Julien, Hiroyuki Iida, and BRUNO BOUZY. "The Monte-Carlo Approach in Amazons". *Computer Games Games Workshop*. 2007.

[29] Knuth, Donald E. and Ronald W. Moore. "An Analysis of Alpha-Beta Pruning". *Artificial Intelligence*, 6:293–326, 1975.

[30] Kocsis, Levente and Csaba Szepesvári. "Bandit based Monte-Carlo Planning". *In: ECML-06. Number 4212 in LNCS*, 282–293. Springer, 2006.

[31] Lee, Chang-Sing, Matrin Müller, and Olivier Teytaud. "Special Issue on Monte Carlo Techniques and Computer Go". *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):225–228, December 2010.

[32] Lorentz, Richard. "Amazons Discover Monte-Carlo". H. Jaap van den Herik (editor), *Computers and Games*, volume 5131, 13–24. Springer Berlin Heidelberg, 2008.

[33] Lorentz, Richard and Therese Horey. "Programming Breakthrough". *8th International Conference on Computers and Games*. Yokohama, Japan, 2013.

[34] Lorenzi, Rossella. "Oldest Gaming Tokens Found in Turkey". URL http://news.discovery.com/history/archaeology/oldest-gaming-tokens-found-130814.htm.

[35] Maarten, Schadd. *Selective Search in Games of Different Complexity*. Ph.D. thesis, University of Maastricht, 2011.

[36] Ramanujan, Raghuram, Ashish Sabharwal, and Bart Selman. "On Adversarial Search Spaces and Sampling-Based Planning". *International Conference on Automated Planning and Scheduling*. 2010.

[37] Rijswijck, J. van. *Computer Hex: Are bees better than fruitflies?* Master's thesis, University of Alberta, Edmonton, Alberta, Fall 2000.

[38] Russell, Stuart and Peter Norvig. *Artificial Intelligence, A Modern Approach*. Pearson Education, Inc., Upper Saddle River, New Jersey 07458, third edition, 2010.

[39] Sackson, Sid. *A Gamut of Games*. Random House, New York, 1969.

[40] Saito, Jahn-Takeshi. *Solving Difficult Game Positions*. Ph.D. thesis, University of Maastricht, 2010.

[41] Samuel, Arthur. "Some Studies in Machine Learning Using the Game of Checkers", January 2011. URL http://www.cs.unm.edu/~terran/downloads/classes/cs529-s11/papers/samuel_1959_B.pdf.

[42] Schaeffer, Jonathan. "History Heuristic and Alpha-Beta Search Enhancements". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, II(II):1203–1212, November 1989.

[43] Schaeffer, Jonathan. "Checkers is Solved". *Science*, 317, September 2007.

[44] Sreedhar, Suhas. "Checkers, Solved!", July 2007. URL http://spectrum.ieee.org/computing/software/checkers-solved.

[45] Tarrant, Carolyn, Mary Dixon-Woods, Andrew Coleman, and Tim Stokes. "Continuity and Trust in Primary Care: A Qualitative Study Informed by Game Theory". *Annals of Family Medicine*, 8(5):440–446, September 2010. URL http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2939420/.

[46] Tesauro, Gerald. "Temporal Difference Learning". *Communications of the ACM*, 38(3):58–68, March 1995.

[47] Thomsen, Thomas. "Lambda-search in game trees – With application to Go". *ICGA*, 23(4):203–217, 2000.

[48] Turing, A. M. "Computing Machinery and Intelligence", November 2012. URL http://www.loebner.net/Prizef/TuringArticle.html.

[49] Wagner, J. and I. Virag. "Solving Renju". *ICGA*, volume 24, 30–34. 2001.

[50] van der Werf, Erik, H. Jaap van den Herik, and Jos W.H.M. Uiterwijk. "Solving Go on Small Boards". *ICGA*, 92–107, June 2003.

[51] Winands, Mark H.M. *Analysis and Implementation of Lines of Action*. Master's thesis, University of Maastricht, August 2000.

[52] Winands, Mark H.M. *Informed Search in Complex Games*. Ph.D. thesis, University of Maastricht, December 2004.

[53] Winands, Mark H.M. "6 x 6 LOA is Solved". *ICGA Journal*, 234–238, December 2008.

[54] Winands, Mark H.M. and Tngvi Bjornsson. "Alpha/Beta Based Play-outs in Monte-Carlo Tree Search". 2011.

[55] Winands, Mark H.M. and Yngvi Björnsson. "Evaluation Function Based Monte-Carlo LOA". *12th International Conference, ACG 2009*, 33–34. May 2009.

[56] Winands, Mark H.M. and Yngvi Björnsson. "Alpha/Beta Based Play-outs in Monte-Carlo Tree Search". *IEEE Conference on Computational Intelligence and AI in Games*, 110–117. 2011.

[57] Winands, Mark H.M., Yngvi Björnsson, and Jahn-Takeshi Saito. "Monte-Carlo Tree Search Solver". *Proceedings of the 6th International Conference on Computers and Games*, 25–36. Springer-Verlag, 2008.

[58] Winands, Mark H.M., Yngvi Björnsson, and Jahn-Takeshi Saito. "Monte Carlo Tree Search in Lines of Action". *IEEE Transactions on Computational Intelligence and AI Games*, 2(4):239–250, December 2010.

[59] Winands, Mark H.M and J.W.H.M. Uiterwijk. *PN, PN2, PN* in Lines of Action*. The CMG Sixth Computer Olympiad: Computer-Games Workshop Proceedings 0922-8721, University of Maastricht, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands, 2001.

[60] Yoshimoto, H., K. Yoshizoe, T. Kanoeko, A. Kishimoto, and K. Taura. "Monte Carlo has a way to go". *Proceedings of the 21st National Conference on Artificial Intelligence*, 1070–1075. 2006.

[61] Ziles, Sandra and Robert Holte. "The computational complexity of avoiding spurious states in state space abstraction". *Artificial Intelligence*, 174:1072–1092, June 2010.

[62] Zobrist, Albert. *A New Hashing Method with Application for Game Playing*. Technical Report 88, The University of Wisconsin, April 1970.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
27–03–2014	Master's Thesis	June 2012–March 2014

4. TITLE AND SUBTITLE

Complexity, Heuristic, and Search Analysis
for the Games of Crossings and Epaminondas

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

King Jr, David W., Captain, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT-ENG-14-M-44

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Intentionally Left Blank

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

13. SUPPLEMENTARY NOTES

This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Games provide fertile research domains for algorithmic research. Often, game research helps solve real-world problems through the testing and refinement of search algorithms in game domains. Other times, game research finds limits for certain algorithms. For example, the game of Go proved intractable for the Min-Max with Alpha-Beta pruning algorithm leading to the popularity of Monte-Carlo based search algorithms. Although effective in Go, and game domains once ruled by Alpha-Beta such as Lines of Action, Monte-Carlo methods appear to have limits too as they fall short in tactical domains such as Hex and Chess. In a continuation of this type of research, two new games, Crossings and Epaminondas, are presented, analyzed and used to test two Monte-Carlo based algorithms: Upper Confidence Bounds applied to Trees (UCT) and Heuristic Guided UCT (HUCT). Results indicate that heuristic knowledge can positively affect UCT's performance in the lower complexity domain of Crossings. However, both agents perform worse in the higher complexity domain of Epaminondas. This identifies Epaminondas as another domain that poses difficulties for Monte Carlo agents.

15. SUBJECT TERMS

Games, Artificial Intelligence, UCT, Crossings, Epaminondas

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			LTC Robert J. McTasney (ENG)
U	U	U	UU	106	19b. TELEPHONE NUMBER *(include area code)* (937) 255-3636 x4460 robert.mctasney@afit.edu

Standard Form 298 (Rev. 8–98)
Prescribed by ANSI Std. Z39.18